
# UNIFIED BEHAVIOR FRAMEWORK FOR REACTIVE ROBOT CONTROL IN REAL-TIME SYSTEMS

THESIS

Brian G. Woolley, First Lieutenant, USAF

AFIT/GCS/ENG/07-11

**DEPARTMENT OF THE AIR FORCE**
**AIR UNIVERSITY**

***AIR FORCE INSTITUTE OF TECHNOLOGY***

**Wright-Patterson Air Force Base, Ohio**

---

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or United States Government.

AFIT/GCS/ENG/07-11

# UNIFIED BEHAVIOR FRAMEWORK FOR REACTIVE ROBOT CONTROL IN REAL-TIME SYSTEMS

THESIS

Presented to the Faculty
Department of Electrical and Computer Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
In Partial Fulfillment of the Requirements for the
Degree of Masters of Science

Brian G. Woolley, B.S.C.E.
First Lieutenant, USAF

March 2007

AFIT/GCS/ENG/07-11

# UNIFIED BEHAVIOR FRAMEWORK FOR REACTIVE ROBOT CONTROL IN REAL-TIME SYSTEMS

Brian G. Woolley, B.S.C.E.

First Lieutenant, USAF

Approved:

| | |
|---|---|
| /signed/ | March 2007 |
| Gilbert L. Peterson, PhD (Chairman) | date |
| /signed/ | March 2007 |
| John F. Raquet, PhD (Member) | date |
| /signed/ | March 2007 |
| Maj Michael Veth, PhD (Member) | date |

## *Abstract*

Endeavors in mobile robotics focus on developing autonomous vehicles that operate in dynamic and uncertain environments. By reducing the need for human-in-the-loop control, unmanned vehicles are utilized to achieve tasks considered dull or dangerous by humans. Because unexpected latency can adversely affect the quality of an autonomous system's operations, which in turn can affect lives and property in the real-world, their ability to detect and handle external events is paramount to providing safe and dependable operation. Behavior-based systems form the basis of autonomous control for many robots. This thesis presents the unified behavior framework, a new and novel approach which incorporates the critical ideas and concepts of the existing reactive controllers in an effort to simplify development without locking the system developer into using any single behavior system. The modular design of the framework is based on modern software engineering principles and only specifies a functional interface for components, leaving the implementation details to the developers. In addition to its use of industry standard techniques in the design of reactive controllers, the unified behavior framework guarantees the responsiveness of routines that are critical to the vehicle's safe operation by allowing individual behaviors to be scheduled by a real-time process controller. The experiments in this thesis demonstrate the ability of the framework to: 1) interchange behavioral components during execution to generate various global behavior attributes; 2) apply genetic programming techniques to automate the discovery of effective structures for a domain that are up to 122 percent better than those crafted by an expert; and 3) leverage real-time scheduling technologies to guarantee the responsiveness of time critical routines regardless of the system's computational load.

## *Acknowledgments*

If the enthusiasm and effort that I put into this research deserves any credit, then so do the people that stood by and helped shape my creative energy into a coherent set of ideas. First and foremost, I must thank my wife for her support and understanding, without her reassurance and encouragement I'd have been consumed by this task. She is truly my partner and my friend. I am better for knowing her and this work has benefited from that fact. To my daughters, who are (and always will be) my source of inspiration and an oasis of sanity. And finally, to Dr. Peterson, your guidance and input has shaped this research and helped me accomplish more than I would have believed myself to be capable of.

*Table of Contents*

Page

## *List of Figures*

## *List of Tables*

# UNIFIED BEHAVIOR FRAMEWORK FOR REACTIVE ROBOT CONTROL IN REAL-TIME SYSTEMS

## I. Introduction

Robots and autonomous vehicles are used to achieve tasks that are dull, dangerous or difficult for humans. Although many robots are, and will continue to be controlled remotely, the need for human-in-the-loop control is shifting towards the assignment of high level objectives. Interplanetary applications, such as the exploration of Mars, struggle with one-way communication latencies between 10 and 20 minutes that make direct control of vehicles impractical. Research at NASA is exploring the use of rovers, airplanes, and balloons for exploration, such systems require the ability to accept broad direction and then conduct operations within the environment without human intervention. For systems operating autonomously in the real-world, an ability to detect and handle external events is paramount to providing safe, predictable, and dependable operation in environments where change and uncertainty is normal.

In addition to the technical challenges presented by the design of autonomous systems, the effort required for development and testing grows exponentially with the addition of new capabilities. Traditionally a mobile robot design implements a specific type of behavior architecture for low-level control and risks becoming platform specific. This thesis proposes that implementing a modular behavior framework that encapsulates the reactive controller's behavioral logic eases the development and testing of low-level routines by encouraging reuse and portability—resulting in faster development cycles that produce behaviors with fewer errors. Further, the modular design of the framework allows developers to use real-time scheduling techniques to ensure that a system's critical routines remain predictably responsive to changes in the environment. In contrast to concurrent programming, the use of real-time technologies eases the design of systems that are both logically and temporally correct by eliminating the potential for unpredictable delays.

This chapter provides a high-level overview of the research conducted in this investigation. It covers the problem to be solved, an overview of behavior-based robotic applications and the need to provide real-time services to guarantee the responsiveness of critical system routines, the goals and objectives of this research investigation, and the sponsors. Chapter I also highlights the assumptions and risks of this research and provides an overview of the thesis document.

## *1.1 Problem Statement*

Autonomous systems that operate in the real-world exist in an unbounded domain and have an inherent requirement to be both robust and responsive when faced with sudden and unpredictable changes in the environment. Behavior-based systems form the basis of autonomous control for many robots, each attempting to strike a balance between rational action and responsiveness. Traditionally, mobile robots implement a single behavior architecture for a given domain, thus binding its capabilities to the strengths and weaknesses of that architecture. This thesis proposes that many of the existing behavior-based approaches share critical aspects and that each can be represented by a single straightforward framework that attempts to: 1) simplify development and testing; 2) promote the reuse of code; 3) support large hierarchical designs while restricting code complexity to base behaviors, and most importantly; 4) allow the developer the freedom to use the behavior-based system they feel is the most appropriate for the given domain.

Additionally, research efforts for the Cooperative Autonomous Navigation and Sensing AFOSR lab task at AFRL/SNR indicate that timing is critical for routines that capture navigational data, especially as a vehicle's frequency of motion increases. Autonomous navigation techniques that blend inertial and external reference data are sensitive to processing delays. Applications that are sensitive to timing errors require the ability to preempt a running process with bounded latency, making the system predictably responsive, even in unpredictable environments. The modular design of the unified behavior framework allows the execution of time critical behavior elements to be managed using real-time scheduling techniques to ensure safe, dependable robot operation in dynamic and uncertain domains by guaranteeing that the routines that detect and handle external events remain responsive.

## *1.2 Key Concepts*

The development of autonomous vehicles that can operate safely in dynamic and uncertain environments requires systems that are not only logically correct, but are predictably responsive as well. This requirement stems from the need for systems to allow deliberative process the computational time to set goals and perform planning operations while ensuring that the reactive routines that provide for the safety of the system are faithfully executed at scheduled intervals. The following is a high-level discussion of behavior-based robotics and the importance of real-time process scheduling.

### *1.2.1 Behavior-based Robotics*

Ideally mobile robotics applications place vehicles into environments where uncertainty is normal. Environments like homes, offices, and public areas are inherently unconstrained, and the use of cost effective motors and sensors that are inaccurate and prone to failure add additional uncertainty about the environment.

As intelligent vehicles are expected to handle increasingly more complex endeavors, their design, implementation and testing requirements grow by orders of magnitude. Symbolic approaches to world modeling are quickly overwhelmed trying to represent large or fine grain environments, both in computational time requirements and memory requirements. Alternatively, strictly reactive behavior-based controller implementations attempt to maintain responsiveness by abandoning goal directed optimality.

Reactive *behavior-based controllers* employ a small number of behaviors using a single arbiter to perform action selection. These designs provide robust low-level control but are customized for specific environments, which makes reuse of these control structures for different scenarios or reuse on different robots rare. Reactive approaches are also plagued by complexity issues when they attempt to scale up to support additional system responsibilities. As theses controllers attempt to deliver reactive behaviors that are increasingly rational and goal oriented, they quickly grow in complexity, necessarily moving away from their roots as robust and responsive control elements. Thus reactive

controllers reach a *capability ceiling* because they lack a mechanism for managing their complexity [33].

To combat the capability ceiling, and because the deliberative qualities of symbolic approaches are equally important as the responsiveness of reactive approaches, both are combined in three-layer architectures [26]. A planning layer sets goals for the system which are then achieved by a sequencing element that activates and deactivates simple behaviors in a temporal order that achieves a higher order goal. The elegance of this approach is that low-level behaviors do not know the goals that they will be used to achieve [23]. In this context, reactive behaviors do not need to know the systems overall goals a priori, which allows them to remain simple, specialized and responsive.

If sequences of specialized behaviors are usable as a means of working towards and achieving higher order goals and plans, then a mechanism exists that allows individual behaviors to be called on or activated interchangeably. The key question is, “How general is this mechanism?” Experience in the design of software systems has established that encapsulation via a well defined interface is an effective means allowing independent components to be used as interchangeable components. A direct result of abstracting specific implementations behind a general interface is that modular components become reusable.

Reusability is an important attribute of a system because it reduces the time required for development and testing. Normally there is a significant amount of redundant control logic in a collection of task-oriented behaviors, thus we ask ourselves, “How can a new behavior be constructed quickly as a simple adaptation or construction of existing elements?” By incorporating existing behavior modules that have been previous tested for correctness, new designs capitalize on reuse as a way of reducing their design complexity. Hence, a hybrid controller or three-layer control architecture that incorporates the unified behavior framework as its behavioral basis enforces a generalized interface that supports modularity, functional abstraction, and reuse. Further, because the framework is modeled after the composite pattern [25], new behaviors can be formed as arbitrated hierarchies of atomic behaviors, existing hierarchical structures or any combination of the two.

The use of a framework that enforces a common interface for all behaviors and allows new behaviors to be formed rapidly as constructions of existing behaviors has two key affects on the design of reactive behavior structures. One is that the complexity of developing robust and coherent behaviors is reduced because atomic behavior elements become highly focused and more easily validated. The other is that the implementation approach of each behavior is left to the developer, allowing the use of a structure that is effective in given situations or simply one that is more familiar.

The ridged use of abstraction to establish functional boundaries creates a modular environment where new structures can be easily formed as arrangements of existing components. The ability to use existing behavior structures in the construction of new ones allows designs to scale easily into large hierarchies while restricting code complexity to the base behaviors. Although many structural combinations will not yield coherent behaviors, experimentation is encouraged to discover ones that are semantically correct. Because individual behavior and arbiter elements are validated independently, the outward attributes of a behavior are isolated from the structural composition. Chapter V presents a case study that uses a genetic program to demonstrate how experimentation is used to automate the discovery of effective structural combinations.

### *1.2.2 Real-Time Scheduling*

The development of robotic systems that attempt to balance their ability to be both deliberative and responsive in dynamic and changing environments face a difficult problem because simple processes that execute at frequent intervals are interleaved with planning and optimization algorithms that need to run for relatively long periods. Such a situation potentially introduces unpredictable delays where high-priority control routines are forced to wait until lower priority planning elements yield or are preempted by the operating system. Ideally, deliberative processes execute “between” the periodic execution of low-level control routines and the amount of computational time available for planning then fluctuates in response to the amount of change in the environment. In chaotic environments a system may be operating under reactive control continuously to maintain a safe operating envelope, leaving little time for deliberation. In quieter environments reactive control is only needed periodically, allowing the remaining

processor time to be used for deliberative calculations. The root of the problem is that the schedulers used by modern operating systems do not guarantee that the highest priority process will be the running process, only that the highest priority process will run next.

The need to make some processes "more important" than others is becoming common in applications where responsiveness is important and milliseconds of delay count. Real-time operating systems are emerging that allow developers to designate some routines as time-sensitive, allowing them to immediately preempt the currently running process. The modular design of the unified behavior framework supports the ability to implement reactive control elements for real-time domains. This approach allows some or all of a system's atomic behaviors to be scheduled as real-time tasks that can preempt the execution of higher-level processes. Section 3.3 provides an example of how real-time scheduling approaches allow periodic processes to be interleaved in a predicable manner that satisfies the needs of the system. Chapter V presents a case study that implements the multithreaded version of the framework that guarantees the periodic execution rate of the base behaviors by running in the context of a real-time operating system, thus demonstrating the ability of a mobile robot to remain responsive regardless of the processing load created by its deliberative ability.

## *1.3 Research Goal*

The overall and guiding goals of this research are: 1) to show that a single straightforward framework can be used to represent many of the existing behavior-based approaches; and 2) to verify that such a framework supports the implementation of behavior-based controllers using real-time systems to ensure the responsiveness of low-level control processes.

## *1.4 Sponsor*

This research is part of the Cooperative Autonomous Navigation and Sensing (CANIS) Air Force Office of Scientific Research (AFOSR) lab task at the Reference Systems Branch of the Air Force Research Laboratory (AFRL/SNRN), Wright-Patterson Air Force Base. The autonomous navigation requirements for CANIS require the development of the real-time control system architecture presented in this thesis. Such a

control structure composes fundamental services used by higher level systems in a real-time sensing and control environment for autonomous operation of unmanned land and aerial vehicles.

## *1.5 Assumptions*

The techniques and approaches presented in this thesis attempt to avoid any requirement that a developer use a specific implementation language. It does, however, advocate the use of design techniques that draw on the fundamental principles of object oriented (OO) programming, which is currently the dominant programming paradigm. This assertion allows for the use of both statically-checked and dynamically-checked languages. The discussions and diagrams related to software implementation assume that the reader has basic exposure to UML notation [51] and OO design patterns [25].

The development of generalized code that is reusable across domains is an intractable problem while the reuse of design models is effective for establishing patterns of development across domains [28]. For this reason, general design concepts are presented that can be applied to encourage reuse and reduce development effort within a specific domain and provide a familiar design model that reduces the complexity of design, development, and testing in new domains.

## *1.6 Thesis Overview*

The structure of this thesis is as follows: Chapter I introduces the problem and research goals. Chapter II provides a thorough overview of behavior-based robotics, presenting the aspects that apply to the ability of such systems to maintain a responsive basis of control for mobile robots. An introduction to concurrent and real-time programming is presented in Chapter III which also explores the current research efforts in mobile robotics that employ real-time architectures to ensure safety via guaranteed responsiveness in dynamic and uncertain domains. Chapter IV presents the detailed design of a unified behavior framework that accommodates the critical characteristics of existing reactive behavior structures, allowing them to be represented and developed as encapsulated modules that share a common interface. Three implementations of the unified behavior framework are presented as a means of highlighting the usage of the

framework to ease the complexity of designing and testing robot behaviors in Chapter V. Each case study is presented as an isolated experiment, including a description of its implementation, the associated results and a discussion of the results as they relate to the use of the framework. The final chapter, Chapter VI, presents concluding remarks and recommendations for future research into the use of real-time systems to enhance the capabilities of behavior-based robotics in changing environments.

# II. Behavior-Based Robotics Background

This chapter presents the current research in the area of behavior-based robotics with a particular focus on the deliberative and reactive aspects that each system contributes. The primary goal in the development of autonomous mobile robot systems is to create a system that can pursue goals and perform useful tasks in dynamic and unpredictable environments. Research in this field has established that such systems must be responsive to changes in the environment while maintaining the deliberative capability to make rational decisions about their actions. This requirement is necessary to ensure the safe and dependable operation of autonomous robots that may work and coexist along side humans or be used for extended periods of time in hostile environments without human intervention.

The background material in this chapter is presented in three sections. The first section discusses traditional efforts to develop rational robots using symbolic world modeling, followed by a discussion of the reactive control architectures. The final section presents the three-layer architecture, the current architectural paradigm for mobile robots.

## *2.1 Symbolic World Modeling*

Research efforts in robotics through about 1985 focused almost exclusively on planning and world modeling [26] in an attempt to develop completely rational mobile robots [44]. This sense-plan-act approach, Figure 2.1a, proved inadequate in dynamic and unpredictable environments, where the robot finds itself in trouble when its internal state loses sync with the reality that it is intended to represent [2]. This is because anything approaching a real world model typically requires so much time to maintain and develop plans for, that the state of the environment changes before the actions can be carried out, effectively nullifying the action sequence. Shakey [41] and Rover [39] are early examples of implementations that depend heavily on symbolic models of the world [16] and subsequently do not perform well in dynamic environments. The main problem is that a traditional Lorenz control loop [48] directly links the rate at which a robot can evaluate

and act on its environment to the computational time requirements of the planning module.

The serial execution of the sense-plan-act control loop makes this paradigm most appropriate for simple robotic systems that do not require complex decomposition of the environment or the problem domain. For complex tasks, a different decomposition approach is needed to maintain a responsive control loop.

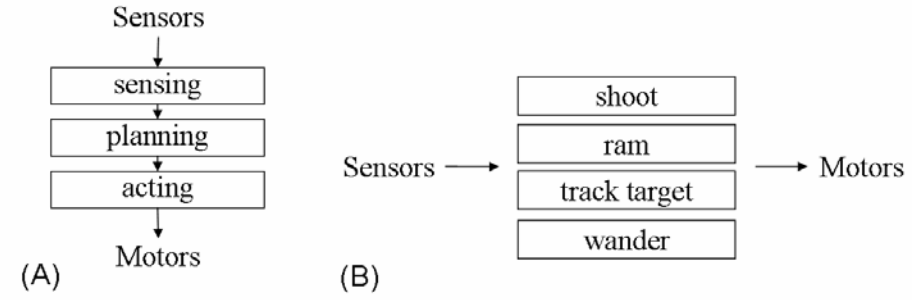



Figure 2.1: Two organizational decomposition strategies for robot control (A) Sequential execution of functional modules; (B) Layered, task-based modules with parallel execution.

## *2.2 Reactive Control Architectures*

The need to alleviate this planning bottleneck led tasks to be decomposed into collections of low-level primitive behaviors, Figure 2.b. This approach takes on the self-contradictory term, *reactive planning* [26]. The ideas behind reactive planning stem from arguments such as Braitenberg's, who argues that the complex behavior of natural organisms may be the result of simple behaviors. Braitenberg further argues that by combining simple behaviors, more complex behaviors and attributes are possible [14]. In equivalent research Brooks claims that for many tasks, robots do not need traditional reasoning, only a tight coupling of sensing to action. He backs that claim with robust autonomous robots using the Subsumption architecture [15, 17].

### *2.2.1 Subsumption.*

The Subsumption architecture advocates for a layered control system based on task decomposition, an approach which is radically different from previous research. Figure 2.1 highlights the quintessential paradigm shift from the sense-plan-act architecture to the new horizontal structure of Subsumption. This parallel organization naturally promotes

concurrent and asynchronous responses to sensor input. Each individual layer works to achieve its particular goal. Coordination between layers is achieved when complex actions (or higher layers) subsume simpler actions, or when low-level behaviors inhibit higher layers.

Following Subsumption other reactive architectures emerge as effective robot control structures, each upholding the original tenants of reactive planning. Specifically, a system must be responsive to the environment, include a tight coupling of sensing to action (keeping little or no state representation), and should be robust, able to perform in the face of unanticipated circumstances or sensor failure. Additionally, systems need to be modular, using incremental development to add capabilities, and the system must have an execution that embraces parallelism and concurrency [4]. Many of these principles are a direct rejection of the monolithic sense-plan-act approach. In addition to Subsumption, five other well known reactive architectures are described here: Motor Schema, Circuit Architecture, Action-Selection, Colony Architecture, and Utility Fusion.

### *2.2.2 Motor Schema*

Unlike the priority-ordered layers of Subsumption, the motor schema architecture emerged as a cooperative control approach, allowing for the simultaneous pursuit of multiple goals. This approach captures behavioral primitives as vector fields that support specific perception tasks (e.g., obstacle avoid, move-to-goal, stay-on-path, etc.) which are arbitrated as a normalized vector summation to form a continuous potential field. Since all schemas contribute to the resultant vector, the overall behavior of the robot is a byproduct of its individual schema goals. This approach is useful in navigation tasks, but is subject to local minima and cyclical paths [2].

### *2.2.3 Circuit Architecture.*

The circuit architecture is a hybridization that allows reactive behavior elements and logical formalisms to be bundled into arbitrated collections. Because priority arbitration occurs at each level of abstraction, developers can bundle unlike approaches into mediated hierarchies that are combinations of reactive approaches, logical formalisms, and situated automata [33].

### *2.2.4 Action-Selection*

Action-selection is an architecture that uses activation levels as a dynamic mechanism of behavior selection. Individual behaviors are grouped as competence modules that respond when predefined conditions are detected. Activation levels are used to indicate a level of confidence that can persist while specific conditions exist or may decay over time. Action coordination is achieved by selecting the competence module with the highest activation level. When used in dynamic environments, action-selection gives the global behavior an emergent quality because there is no predefined layering or order of execution. Based on events in the environment, a robot may suddenly begin to display radically different attributes [37].

### *2.2.5 Colony Architecture*

Colony architecture is a direct descendant of Subsumption, allowing higher layers to suppress lower layers but eliminates the ability of lower layers to inhibit higher ones. As a result of enacting the suppression only approach, the colony architecture breaks away from the total ordering of layers found in Subsumption and permits the formation of priority based behavior hierarchies [20].

### *2.2.6 Utility Fusion*

The utility fusion architecture is an expansion of DAMN (Distributed Architecture for Mobile Navigation) [43]. Under this architecture, action selection is coordinated via an evaluation of the utility that would result from taking a particular action from a discreet set of actions. Like DAMN, the arbiter is central to the architecture, taking on the unique kinematics of the specific robot, allowing the evaluation behaviors to remain platform independent and reusable. Behaviors use their own criteria to assess the utility of a proposed future state. The action that collects the highest overall utility is enacted by the arbiter. Although actions are enacted in a winner-takes-all fashion, the utility fusion approach is considered to be cooperative because it selects the action that best serves the global goals of the system [44].

### *2.2.7 Limitations of Reactive Control*

Most behavior-based controllers employ a single reactive architecture as a basis of control. While these designs provide robust low-level control, they are customized for specific environments and limited to the capabilities of the chosen architecture. The coordination of a broad set of behavioral skills to achieve a coherent complex behavior is an error-prone and tedious task that seems to be more of an art than a science [52] , which makes the reuse of existing control structures and behaviors on different robots uncommon. Such reactive approaches are further plagued by complexity issues when they attempt to scale up to support additional system responsibilities. As controllers attempt to deliver reactive behaviors that are increasingly rational and goal oriented, they quickly grow in complexity, necessarily moving away from their roots as robust and responsive control elements. Eventually, reactive controllers reach a *capability ceiling* because they lack a mechanism for managing their complexity [15].

The introduction of the BAP (*behavior, action pattern, policy*) framework breaches the subject of the capability ceiling faced by behavior-based controllers, asserting that the current architectures seem to have been established without much attention to modern software engineering techniques that are the industry standards of application programming [52]. In his paper, Utz suggests that any general purpose behavior architecture must address three key questions:

1) How well will this behavior hierarchy scale to high levels?

2) Does it allow for the reuse of behaviors?

3) Can behaviors and planning be integrated?

In addition to supporting the original tenants of reactive control, a general-purpose architecture should: maintain concurrent behavior execution as indispensable, support the ability to use multiple arbitration mechanisms, allow for the temporal sequencing of behavior sets to provide easy taskability of the robot, provide behavioral parameterization via functional abstraction, and support a hierarchical building policy with proper entry and exit semantics [52].

Limitations of the sense-plan-act architectures to be responsive brought the advent of reactive control architectures that tightly couple sensing with action. Similarly, the need for reactive approaches to be deliberative resulted in the three-layer architecture.

## *2.3 Three-Layer Architectures*

While reactive architectures that are organized as task based decompositions are responsive and able to operate in dynamic environments, they forfeit the ability to make plans and pursue goals. Because deliberative planning and reactive control are equally important for mobile robot navigation, when used appropriately, each complements the other and compensates for the other's deficiencies [44].

Driven by the requirement that systems must not only be responsive in dynamic environments but rational and deliberative as well, the three-layer architecture has become a common paradigm for designing autonomous robot control architectures [11, 26]. Under this approach, the structure of the software system consists of three main components: a reactive feedback control mechanism (the controller), a slow deliberative planner (the deliberator), and a sequencing mechanism that connects the first two components (the sequencer). Each layer of the architecture provides additional environment and sensor abstraction over the previous, and focuses on larger reasoning and goal time scales.

The three-layer architecture's incorporation of previous reactive planning approaches to form the controller element becomes known as *reactive execution* [26] where primitive behaviors are used to deliver robust and responsive low-level control. Despite the previous work on reactive behaviors, the controller remains the most time consuming segment to design and implement. Two three-layer architectures are discussed in detail, namely the Saphira architecture [34] and the 3T architecture [12].

### *2.3.1 Saphira Architecture*

The Saphira architecture is an integrated sensing and control system for robotic applications, which is implemented on Flakey, a custom research robot, and Pioneer, a small commercial robot from Real World Interface. While reactive behavior-based approaches are used to accomplish low-level control, a geometric representation of the

space around the robot is also kept that mediates between perception and action. This central representation, referred to as the Local Perceptual Space (LPS), is maintained by perceptual routines and used by action routines. At first glance, Saphira appears to be using a standard Blackboard architecture [48] where routines interact via a shared information space. Rather, the organization of Saphira is partly vertical and partly horizontal, using independent tasks to perform sensor fusion to update the LPS, which then affects the decisions of the action routines [34]. The vertical element emerges from the organization of perceptual and action routines into levels of cognitive ability.

### *2.3.2 Three Tiered Robot Control*

The three tiered (3T) robot control architecture has been in use at NASA Johnson Space Center since 1992 in a variety of space robot research [13]. In general, 3T separates the basic robot intelligence problem into three interacting pieces, with each subsequent piece being increasingly rational. First, there is a layer of reactive skills that are robot specific. Skills at this layer are fast feedback loops that tightly couple sensing to acting. The next layer is a sequencing element which activates reactive skills in an order that moves the world state towards the current goal set by the planner. The sequencing portion of the architecture uses the Reactive Action Packages (RAPs) system [23] to organize and execute chains of procedures. At the highest level, Prodigy [53] is used as the deliberative planning and learning element that reasons in depth about goals, resources and timing constraints.

## *2.4 Summary*

Typically, the deliberative algorithms and world representations that deliver rational plans and optimal solutions require considerable computation time and are unsuitable when used in dynamic environments. At the other end of the spectrum are simple algorithms that tightly couple sensors to motors and provide very fast responses in changing environments, but retain little or no state information and are not capable of planning or achieving long term goals. Each approach has strengths and weaknesses and the goal is to balance the tradeoffs between the system's ability to deliberate and its ability to respond reactively to unexpected changes in the environment. The three-layer

architecture attempts to strike such a balance, delivering the planning ability via a deliberator while maintaining robust low-level control via a reactive execution element.

Despite the use of concurrent execution techniques, implementations of the Saphira and 3T architectures are unable to guarantee that their low-level control processes will execute at regular intervals. Outwardly, the problem is that the robot can become unresponsive while performing time intensive tasks. The root problem is that the independent routines are subject to the underlying scheduling approach used by the base operating system. The scheduling algorithms used by modern operating systems attempt to maximize average performance, which can allow critical routines to be starved by computationally heavy tasks. Chapter III covers current research in robot architectures that use real-time control facilities to guarantee that routines that ensure safe, dependable operation execute at their intended frequency.

# III. Concurrency and Real-Time Robot Architectures

This chapter presents an introduction to concurrent and real-time programming and explores the current research efforts in mobile robotics that employ real-time techniques to ensure safe operation in dynamic and uncertain environments. Section 3.1 provides relevant background on the complexities that are unique to concurrent programming. Section 3.2 presents a discussion of scheduling theories and the current development in real-time operating systems and programming languages. Current research efforts in mobile robotics based on real-time systems is presented in section 3.3. The final section, 3.5, reiterates the importance of providing a guarantee that critical routines will run at specified times and intervals to maintain safety.

## *3.1 Concurrent Programming*

To maintain the responsiveness of a computer, modern operating systems like Windows and UNIX use concurrency to give the appearance that multiple tasks are being handled simultaneously. While tasks appear to be running at the same time, they are actually taking turns, each process interleaving its incremental progress with the progress of the other active processes. Modern operating systems attempt to model parallelism by giving processes *time slices* in which to perform a task before being preempted or forced to yield to the next processes that is ready to run. This approach attempts to maximize "average" performance [58] and provide the appearance of multitasking. In some cases, the overall performance can appear to be quite poor, because the scheduling algorithm gives no assurance about when a process will be allowed to run or that the highest priority task will always be active [54].

Historically, the term *process* was introduced to describe the sequence of actions performed by the execution of a sequence of instructions. Thus a concurrent process is one that can be performed independent of (and possibly at the same time as) another process. Through the use of protected memory space and context switching, operating systems can run multiple independent processes concurrently by interleaving their time slices. More recently, processes have become known as *programs*, because modern

operating systems allow the sub-processes created in the same program (or parent process) to have unrestricted access to the program's memory space; these concurrent processes within a program are known as *threads*. Operating systems that support multithreading have an ability to interleave the execution of active threads by giving each one its own time slice to run in.

Since threads operate within the shared memory space of a program, they are typically required to synchronize and coordinate at critical junctions to achieve their goals and maintain the program's coherent operation. If not implemented properly, concurrent programs introduce the potential for new problems that do not exist in their sequential counterparts [54]. Some typical error conditions are *deadlock*, *interference*, and *starvation*. Deadlock refers to a condition where two or more processes are each waiting for another to release a resource. Interference may occur when two or more concurrent activities attempt to update the same object, resulting in a corruption of the data. Starvation occurs when one or more concurrent activities are perpetually denied resources required to finish a task.

The ability of a thread to maintain a coherent state is typically indicated by the level of thread safety [9] that it supports. Bloch suggests the following levels: *Immutable* instances of a class are constants and cannot be changed, and thus there are no thread safety issues. *Thread-safe* instances of a class are mutable but handle all synchronization internally and can be used safely in a concurrent environment. *Conditionally thread-safe* instances of a class have some methods that are thread-safe or have methods that must be called in sequence. *Thread-compatible* instances of a class provide no internal synchronization and locking, but can be used safely in concurrent environments if the calling process provides the appropriate locks. *Thread-hostile* instances of a class are unsafe to use in concurrent applications [54].

Thread-safe implementations are able to maintain coherent operation in concurrent programs by employing standard communication and synchronization patterns. Some typical ones are: *semaphores*, *signals*, *events*, *reader/writer buffers*, *blackboards*, *broadcasts*, and *barriers*. Semaphores can be either blocking or not-blocking, but are essentially counters that are used to control the number of processes accessing a limited

resource. Threads acquire and release the associated resource through the semaphore. Signals, whether persistent or transient, are used to communicate between threads as a means of synchronizing their progress. Persistent signals remain set until the waiting thread receives it. Transient signals are pulses that are used to release one or more waiting threads. Events are essentially signals that have a specific values, a process waiting on a specific event will unblock when the event occurs. Buffers are typically used to pass messages between threads, and once read the data is destroyed. If the data is to be retained, then the blackboard abstraction is more appropriate. Broadcasts are essentially pulses that pass data to the recipients. Finally, barriers are used to synchronize the execution of threads by block their execution as they arrive at the barrier and then releasing them all once all the registered threads have arrived at the barrier [50, 54].

Despite the complexity introduced over sequential programming, concurrent programming allows programs to remain responsive to user input while tasks are completed as background processes and in most cases provide the appearance of parallelism. Currently, modern operating systems and programming languages allow threads to be interrupted synchronously. While this approach simplifies the control and synchronization requirements, it is inherently weak because it does not provide a guarantee that the interrupted thread will yield in a known period of time. For real-time applications, the ability to asynchronously preempt a running process with bounded latency is a critical element that makes real-time systems predictably responsive, even in unpredictable environments.

## *3.2 Real-Time Systems*

Real-time systems are typically used to control critical systems where an untimely response to an event in the real-world is either too late or incorrect and risks the safety of the public, personnel, or the system itself. The software must, therefore, be engineered to the highest standard, and programs must attempt to tolerate faults and continue to operate, even at degraded levels [54].

A system is said to be real-time if the correctness of an operation depends not only upon its logical correctness, but also upon the time at which it is performed. Such systems provide control facilities that enable a programmer to specify times at which

actions are to be performed or times at which actions are to be completed, as well as the ability to respond or dynamically reschedule tasks when a timing requirement cannot be met. It is also common to distinguish between *hard* and *soft* real-time systems. Hard real-time systems typically have a strict schedule in which processes must complete their task, or forfeit the integrity of the system. This approach is typically implemented as an embedded system and guarantees response times less than the maximum stated latency (normally between 10 and 100 μs). Approaches that can tolerate some lateness are referred to as soft real-time and are typically responsive but can not assert their maximum latency. The violation of timing constraints in soft real-time systems results in degraded quality, but does not necessarily lead to a failure state. In the context of mobile robotics, a system that takes a little longer to make a plan is more acceptable than one that strikes a wall or a researcher because it is too busy making plans.

Although many efforts exist to develop general purpose real-time operating systems, the advent of the POSIX-1003.1b real-time extensions [32], provides UNIX a chance to become the real-time processing platform of choice. Thus, we explore the two main efforts to develop Linux into an operating system capable of meeting hard real-time constraints: RTLinux [58] and RTAI (*Real-Time Application Interface*) [57]. On a separate front, the Java programming language, which currently provides exceptional support for concurrent programming, is being extended to support the development of real-time applications. This effort, sparked by guiding principles set forth by the U.S. National Institute of Standards and Technology (NIST), is known as the Real-Time Specification for Java (RTSJ) [10] and promises to provide solid support for real-time applications programming.

### *3.2.1 Real-Time Linux*

The Real-Time Linux (or RTLinux) development uses an approach known as preemption improvement to shorten interrupt servicing latencies down to levels that support real-time applications [6]. In the preemption improvement approach, the Linux kernel is modified to reduce the length of the longest section of non-preemptible code in order to minimize the latency of interrupts or real-time task scheduling in the system. This is critical because the amount of time spent in the longest section of non-preemptible

code is the shortest scheduling latency that can be guaranteed for a real-time application whose operation relies on specified latency limits to ensure correctness [7].

Use of the preemption improvement approach creates several drawbacks. The first is that any guarantee of maximum latency is effectively unverifiable. Although the kernel is generally more preemptible, such a guarantee is limited unless every possible code path in the kernel is examined. Another limitation is that future maintenance is difficult. The significant deviation that RTLinux takes from the main line of Linux development establishes a new operating system that is unsupportable by the main Linux community. Finally, the preemption improvement approach requires substantial modifications throughout the Linux kernel, which poses the risk of introducing new bugs [8].

### *3.2.2 Real-Time Application Interface (RTAI)*

In contrast to RTLinux, the RTAI development effort uses an approach known as interrupt abstraction to reduce interrupt latency for real-time applications. Instead of making incremental changes to the kernel to improve its preemptibility, RTAI adds a small real-time kernel below the standard Linux kernel and treats the Linux kernel as a low priority real-time task [45]. The Linux kernel runs as RTAI's idle process, only running when there are no real-time tasks to run and the kernel is preempted whenever a real-time task needs to run [6, 58]. Because a separate hardware handling layer intercepts and manages the actual hardware interrupts, any missed hardware inputs are simulated, making the Linux kernel mostly unaware that it is being subverted by RTAI [8].

The interrupt abstraction approach leaves the Linux kernel largely untouched, avoiding many of the software maintenance problems faced by RT-Linux. Additionally, the RTAI scheduler and hardware abstraction layer total 64 kilobytes, which is small enough that it no longer makes verification of the latency guarantees prohibitive [8]. The main draw back to RTAI is that real-time processes are implemented as standalone kernel processes. As of release 1.02a, RTAI supports inter-process communication methods and a symmetrical API that allows real-time tasks to be created from inside the Linux user space, allowing an application to operate using a mixture of real-time and non-real-time tasks [45].

## *3.3 Process Scheduling*

Real-time systems differ from typical computational systems in that they must be able to interact with their environment in a timely and predictable manner, the days when *real-time* simply meant *fast* are long gone [54]. The need to make some processes "more important" than others is becoming common in applications where responsiveness is important and milliseconds of delay count. Consider a time critical process ***A***, when ***A*** is able to run, it should run in place of any other process. This seems intuitive, but the scheduling algorithms used by modern operating systems do not guarantee when ***A*** will become the running process, only that it will be the next process to run.

The two scheduling algorithms defined for real-time operating systems are round-robin and FIFO (first in first out). Both schedulers allow a higher priority process to preempt currently running process at any time. The primary difference between the two real-time schedulers is that FIFO allows a process to run indefinitely while round-robin forces a process to yield and allow another process to run. The schedulers used in modern operating systems attempt to give all processes a "fair share" of the processor and do not observe any specific time constraints.

An example designed to demonstrate the difficulties in the scheduling approaches is used to describe the differences in the way that real-time systems and modern operating systems will schedule an identical processes load. The test load consists of three processes, defined in Table 3.1, where: process ***A*** is a task that occurs at frequent intervals; process ***B*** occurs less frequently but takes longer to complete; while process ***C*** is considerably complex but runs infrequently.

Table 3.1: Process Scheduling Example

| Process | Periodic Execution | | Execution |
|---|---|---|---|
| | *Time* | *Frequency* | Time/Sec |
| A | 3 ms | 20 ms | 150 ms |
| B | 25 ms | 100 ms | 250 ms |
| C | 100 ms | 1000 ms | 100 ms |
| | | Total | 500 ms |

Assuming that the priority of ***A*** is greater than ***B*** and the priority of ***B*** is greater than ***C***, Figure 3.1a shows the execution pattern of a real-time system capable of preempting lower priority processes within 10 µs. The use of a real-time scheduler provides an

asynchronous ability to interrupt the execution of a lower priority process, thus achieving the periodic execution frequencies specified in Table 3.1. In Figure 3.1a, process ***A*** is able to execute in 20 ms intervals by preempting the execution of process ***B***. Likewise, the long (100 ms) execution of process ***C*** is interrupted multiple times to allow both ***A*** and ***B*** to run at their designated intervals.

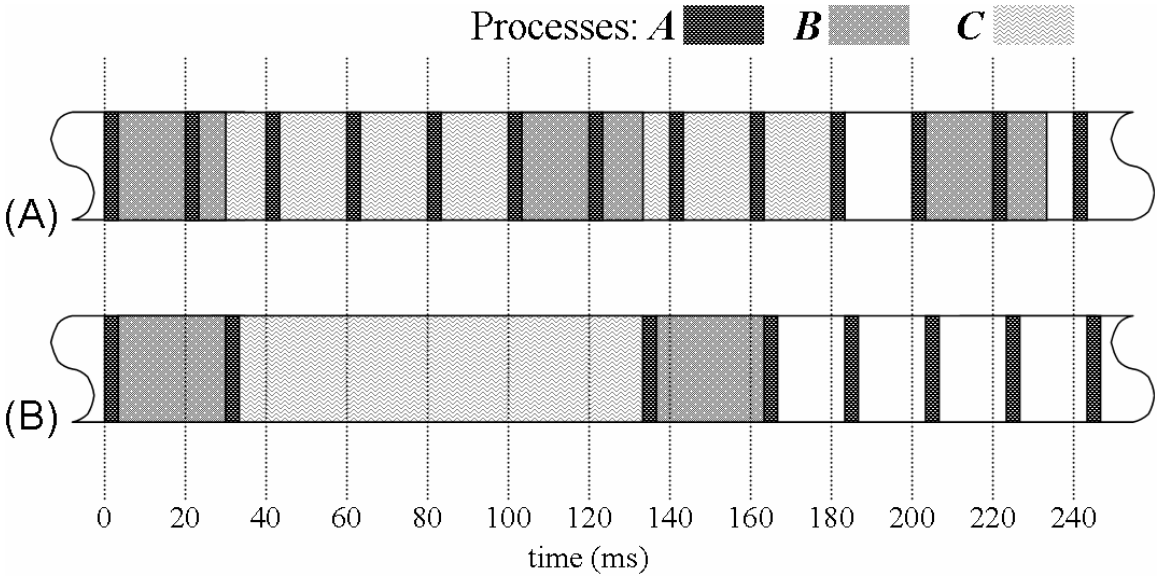



Figure 3.1: Process scheduling timeline for (A) Real-time system with 10 µs preemption; (B) Modern operating system with priority threading scheduling and 100 ms time slices.

While modern operating systems support prioritized thread scheduling, they do not provide fine grain preemption of running processes beyond the default time slice (typically set as 100 ms). A higher priority process that becomes ready to run must either wait for the running process to either yield voluntarily or wait for the operating system to preempt the running process at the end of its time slice. Using the process specifications in Table 3.1, the execution pattern that a modern multithreaded operating system is likely to produce is shown in Figure 3.1b.

Figure 3.1 shows the failure of this scheduling process to meet the periodic execution requirements. Initially, all three processes begin in a ready to run state. The scheduler selects process ***A*** to run followed by process ***B***. During the execution of ***B***, ***A*** renters a ready to run state and because of its higher priority, ***A*** is selected to run ahead of ***C*** and is 8 ms behind schedule. When ***A*** completes its second execution ***C*** is the only process ready to run and enters execution. Because the computation time required by ***C*** is

equal to the operating system's default time slice, it executes uninterrupted for 100 ms before yielding the processor. When **C** completes, process ***A*** is next in line to run and has missed four execution periods and is 83 ms behind schedule. Similarly, the second execution of ***B*** is 31 ms behind schedule. This pattern of disruption repeats at every second due to the periodic execution of ***C***.

This example demonstrates the limits of a priority thread scheduler to provide accurate periodic task execution under heavy computational loads. The following section discusses three robot architectures that use real-time implementations to improve the responsiveness of time-critical routines over long-running deliberative tasks.

## *3.4 Real-Time Robot Architectures*

The three-layer architecture presents a system that is both deliberative and reactive. However, mobile robots exist in the real world where time and events occur continuously and not in discrete time steps. Despite the concurrent execution of each layer, there are no real-time guarantees that the reactive elements providing for the safe operation of the robot will execute as scheduled. The following sub-sections discuss current robot architectures that use real-time approaches to enhance responsiveness and ensure safety.

### *3.4.1 OpenR*

Developed as an open architecture (or multi-vendor system) for autonomous robot systems, OpenR is based on Aperios [59], an object-oriented, distributed operating system which allows physical and software components to be defined uniformly as objects. Because everything is referenced as an object, OpenR advocates for a common interface for various components like sensors and actuators. Expanding on this approach, the design is a layered model consisting of: a hardware adaptation layer (HAL), a system service layer (SSL), and an application layer (APL) [24]. This approach is intended to allow developers to use well defined interfaces and introduce new programs without affecting adjacent layers. Its major weakness is that the HAL layer providing designated services is not sufficiently modular, and thus is not easily enhanced. Another weakness is that OpenR uses message passing to communicate, causing it to suffer long delays that result from messages setting off a cascade effect that results in long service periods prior

to a task being achieved. Though lauded as a real-time system, this approach fails to enforce real-time constraints on process execution and provides no guarantee that a higher priority process will be given access in a timely manner.

### *3.4.2 Miro*

Miro is a CORBA-based robot programming framework [21] intended to allow for the development of reliable and safe robotic software on heterogeneous computer networks and supports the use of several programming languages. The decision to use CORBA supports a common interface wrapper that allows for distributed processing and platform independent code reuse. However the overhead that using CORBA wrappers brings is not conducive to maintaining the responsiveness required by low-level robot control elements. Although the B21 robot implementation was able to accept and schedule tasks from multiple remote workstations it is unclear how the internal robot control was implemented or how that implementation affects responsiveness of the low-level control elements.

### *3.4.3 SmartSoft and OROCOS*

The goal of the SmartSoft [47] and OROCOS [46] projects is to establish robot control frameworks that are both modular and responsive to events in real-time. The central approach to responsiveness is based on the observer pattern [25] which allows a collection of interested components to be immediately notified of an external event. While this approach achieves good results overall, it does not limit the length of the code path triggered by an event, and subsequently cannot guarantee that the system's will remain predictably responsive.

### *3.4.4 YARA*

The YARA architecture [18], which stands for "yet another robot architecture," is unique in that it uses dynamic priority assignment of its activity threads to achieve a responsive basis of control in a changing environment. To improve the dependability of the system and ensure a fast response to environment changes, the priority of each thread of control is tuned to achieve a better coexistence of reactive and deliberative components in the same platform. By adjusting the priority of the activity threads using

an *earliest deadline first* approach, the soft real-time process scheduler available with Linux versions 2.6 or later gives the next time slice to the highest priority thread waiting to run. This approach demonstrates the ability of a general purpose operating system to provide interprocess communication with an average response of 175 µs under optimal conditions. The ability of the YARA architecture to remain stable and predictable under an increasing computational load is demonstrated by implementing two edge following behaviors, one in YARA and another and in SmartSoft [47], a CORBA-based framework partially developed in the context of the OROCOS [46] project and capable of producing 786 µs response times between processes. A major problem exposed by this experiment is also that execution failures went undetected because the SmartSoft architecture had no internal monitoring mechanism to detect processes that failed to execute as scheduled [18]. By dynamically adjusting the priority of the active processes, the improved Linux scheduler will run the highest priority process in the next time slice, but no guarantees are made about responsiveness that are better than the system's established ability to preempt the running process, see section 3.3 and [1].The YARA paper closes by suggesting that hard real-time approaches be explored to improve responsiveness and provide guarantees at fine grain timing intervals.

This thesis expands on the goals of the YARA project by presenting a responsive behavior-based controller design that operates as a collection of periodic tasks managed by a hard real-time scheduler. The assurance that periodic tasks will execute with bounded latency allows the time-critical routines that update and evaluate a shared state to be scheduled at independent intervals while maintaining the stability of the system.

## *3.5 Summary*

Autonomous vehicles and robot architectures are ultimately intended for use in the real-world and therefore must remain responsive to changes in the environment. YARA demonstrates this need for responsiveness, suggesting that the ability of low-level control routines to execute at predictable periodic intervals contributes to a robot's safe and dependable operation [18]. Despite the ability of modern operating systems to support prioritized thread scheduling, these schedulers do not guarantee when the highest priority process will become the running process, only that it will be the next process to run.

Research into general purpose real-time operating systems allows the developer to designate particular threads of execution as real-time processes, effectively bounding the responsiveness of low-level control routines and establishing and ability to schedule tasks at predictable intervals.

# IV. Unified Behavior Framework

This chapter introduces the unified behavior framework (UBF) for reactive control, demonstrates how existing reactive execution architectures can be designed using the UBF and presents two standard implementation approaches. Section 4.1 presents the motivations for establishing a UBF. This is followed by a discussion on the encapsulation of behavioral logic and how a robot controller uses a behavior package. The ability to construct complex behavior structures from existing behaviors is discussed in Section 4.3. The roles of the state and action interfaces are thoroughly presented in Section 4.4 followed by a demonstration of how six well known reactive behavior architectures can be represented in the context of the UBF. Sections 4.6 and 4.7 present implementation examples for sequential, concurrent and real-time domains. The final section reiterates the goals and objectives that the UBF supports.

## *4.1 Purpose*

Traditionally, a mobile robot design implements a single behavior architecture, thus binding its performance to the strengths and weaknesses of that architecture. This section introduces the unified behavior framework which allows a robot to seamlessly change between disparate architectures and provides mechanisms that simplify the design, development and implementation of reactive control structures.

While the purpose of the reactive controller in a three-layer architecture is to form a responsive basis of reactive-control for a mobile robot, a separation can still be made between the controller and the reactive behavior logic. Typically, a robot's low-level controller and its reactive-behavior architecture are developed on an as needed basis, customized for the intended application and expected environment. The creation of a monolithic control structure necessarily ties the controller's behavior to the specific platform, thwarting reuse. Further, the ability to change or expand the base behavior is made difficult because it is developed as an integral part of the controller, making one indistinguishable from the other. By making a clear separation between these two pieces, as see in Figure 4.1, the controller can use various packages of behavior logic without

changing the controller implementation. Additionally, behavior packages are no longer tied to specific platform implementations, encouraging reuse.

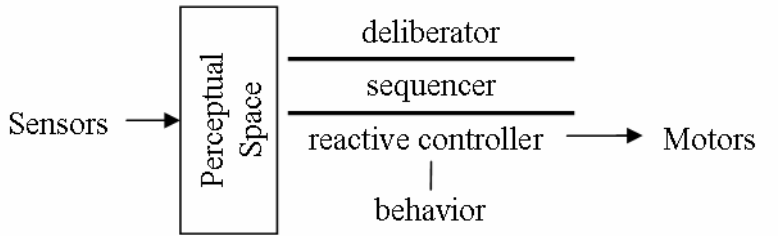



Figure 4.1: Three-layer architecture with deliberate encapsulation of a controller's behavior logic.

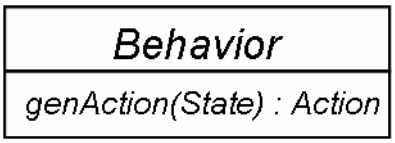



Figure 4.2: Defining an abstract class allows behavior logic to be encapsulated as independent and interchangeable objects that conform to a standard interface.

The UBF uses the strategy pattern [25] to provide the controller with the ability to dynamically swap its behavior packages at runtime. An abstract behavior interface is presented in Figure 4.2 and used to define a family of related algorithms that can be used interchangeably. The controller, knowing how to use a behavior in its abstract form, is able to use any of the concrete implementations that belong to the family of behaviors in a uniform manner, making them fully interchangeable. This approach frees the low-level controller from being bound to any single behavior architecture. In fact it provides the ability to seamlessly switch between distinct architectures during execution. This is advantageous because it promotes the reuse of existing behaviors and frees the developer from being bound to any single behavior architecture.

While the UBF supports code reuse by capturing behavior logic as interchangeable modules, the reuse of subcomponents is also encouraged in the UBF via a mechanism modeled on the composite pattern [25]. The composite pattern allows new control structures to be formed as arbitrated hierarchies of existing behaviors, with the resulting structure being usable as a behavior. The consequence of this is that two or more existing behaviors can be combined (regardless of their underlying architecture) to form a new behavior structure.

The software design mechanisms of the strategy and composite patterns encourage a developer to use modular approaches that ease the complexity of designing, testing and implementing a collection of reactive behaviors, while providing the ability to form larger hierarchies of behaviors. This isolates code complexity to the atomic (or leaf) behaviors. The freedom to join existing behaviors as compositions encourages experimentation with various structural arrangements of elemental behaviors, arbitration components, as well as existing behavior structures. Because the operational logic of subcomponents is independently verified, this approach isolates the task of problem solving a system's outward behavior to the structural arrangement of control modules. While the UBF does ensure that individual elements are compatible with one another, it makes no assertion about the coherence of the resulting control structure. The results in section 5.1 demonstrate these concepts as a robot simulation is used to observe the radical differences that arise in the external behavior attributes as various arbiters are used on an identical set of base behaviors.

## *4.2 Encapsulating Behaviors*

To integrate the UBF into a robot controller, a layer of abstraction is required to make a clear delineation between the controller and the reactive behaviors that drives it. This concept is shown in Figure 4.3, which depicts the control layer as responsible for issuing motor commands based on the recommendations of the active behavior component. The critical aspect is that the behavior modules do not issue commands to the motors, rather they evaluate the shared perceptual space and return a set of recommended motor commands, which are then applies to the robot by the controller. The intent is for the UBF to capture the behavioral logic of the controller as modules that have a consistent interface, and there by provide the ability to seamlessly swap the active behaviors at runtime and provide a responsive and flexible basis of control. Figure 4.3 also expresses that a controller can select its active behavior from a set (or library) of behaviors. In some cases, several behaviors are actively evaluating the perceptual space and making action recommendations. To avoid contention, individual sub-behaviors have no ability to unilaterally enact motor commands on the robot and must make their recommendations via a proxy, see section 4.4 for more about the *Action* interface.

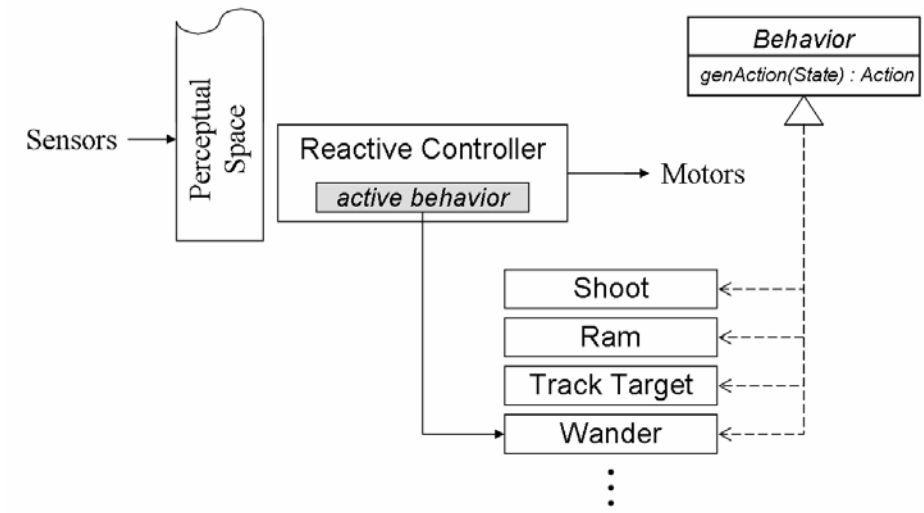



Figure 4.3: Strong encapsulation of reactive behaviors allows the controller to change its active behavior during execution.

Drawing a strict boundary around the behavior logic with a well defined interface allows individual behaviors to be developed that can be used by the controller without impacting its design. Like the reactive action package (RAP) approach developed by Firby [22], the ability to change behaviors during execution allows the temporal sequencing of specialized behaviors to pursue higher order goals without the reactive behaviors requiring information about the plans that they are used to achieve. From an implementation perspective, specialized behaviors have limited complexity and are easier to design and test over monolithic implementations that attempt to address all possible world conditions. Instead, a system that relies on a collection of specialized behaviors with a common interface is able to observe the environmental conditions and apply a particular behavior when it is most effective [30].

Keeping to the fundamental rule that reactive behaviors tightly couple sensing to action suggests that an effective interface must allow sensor inputs and motor command outputs. Normally behaviors are allowed direct access to the sensor and motor hardware. Instead, this is the responsibility of the controller and the behaviors are provided a generic sensor interface referred to as the *State* or Perceptual Space [34] and a generic motor interface referred to as an *Action*.

Using these guidelines, a standardized behavior interface is established that allows an action recommendation to be generated based on the current state. To ensure that all

behaviors have this capability and that they can be used interchangeably, the abstract behavior class, shown in Figure 4.2, is established to serve as the notional definition of all behaviors. Its only requirement is that a *genAction* method be implemented that accepts the state to be evaluated and returns an action recommendation. The creation of a concrete behavior that sub-classes the abstract behavior has two distinct advantages over standalone implementations. The first is that polymorphism requires the presence of a *genAction* method. The second is that all such behaviors are interchangeable because, notionally, they are all behaviors. This is the root concept of the strategy pattern [25].

In order to discuss how the controller might employ this construct, assume that fully implemented behaviors are available. From the controller's perspective, a three-step process is enacted as a continuous loop that is shown by the sequence diagram [51] in Figure 4.4. First, the state is updated to represent the current conditions. Second, the behavior is asked to generate a recommended action by invoking the *genAction* method. Finally, the proposed action is given the authority to issue commands directly to the motors via the *execute* method.

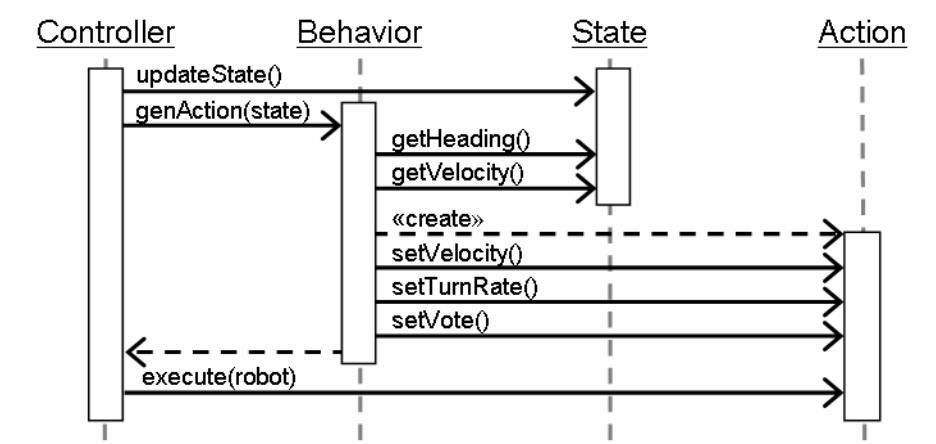



Figure 4.4: Sequence diagram of a controller using its behavior.

In some cases it may be more appropriate to assume that the *State* is always current and relocate this responsibility to an external, asynchronous process that continuously maintains the *State*.

The encapsulation of behavior logic is intended to allow the controller to use disparate behavior-based systems (e.g. Subsumption, action-selection, or utility fusion) as

interchangeable elements of a behavior library. Because a significant goal of the UBF is to encourage the development of reactive control structures that are reusable, interchangeable, and scalable [52], the following subsection continues the discussion about how existing behavior modules are used as the building blocks in the development of more complex behaviors.

## *4.3 Constructing Behaviors*

To support software reuse, the developer requires the ability use existing behavior modules “as-is”, without modification, to form a new reactive controller. The UBF supports this by allowing behaviors to be joined via an arbitration node. Modeled on the composite pattern [25], the arbiter provides the UBF the ability to form hierarchical structures of behavior collections. Thus, a developer is free to reuse the functionality of an existing behavior and incorporate it as a part of a new structure, regardless of its underlying implementation. Now, not only can a controller switch between behavior architectures at will, it can also use a behavior that is a composition of disparate architectures.

The final structure of UBF is presented in Figure 4.5. This class diagram [51] extends the abstract *Behavior* class, adding an arbitrated *Composite* behavior and a *Leaf* behavior. Each composite node is associated with an external *Arbiter*, which allows each joining node in a hierarchy to employ an arbitration technique for the behaviors it groups.

The introduction of the *Leaf* behavior is functionally empty, but serves to identify the atomic building block behaviors of more complex structures. The term “leaf” stems from software structures that are organized hierarchically, as are the terms “tree”, “root”, “branch”, etc. which are useful when discussing a UBF behavior structure.

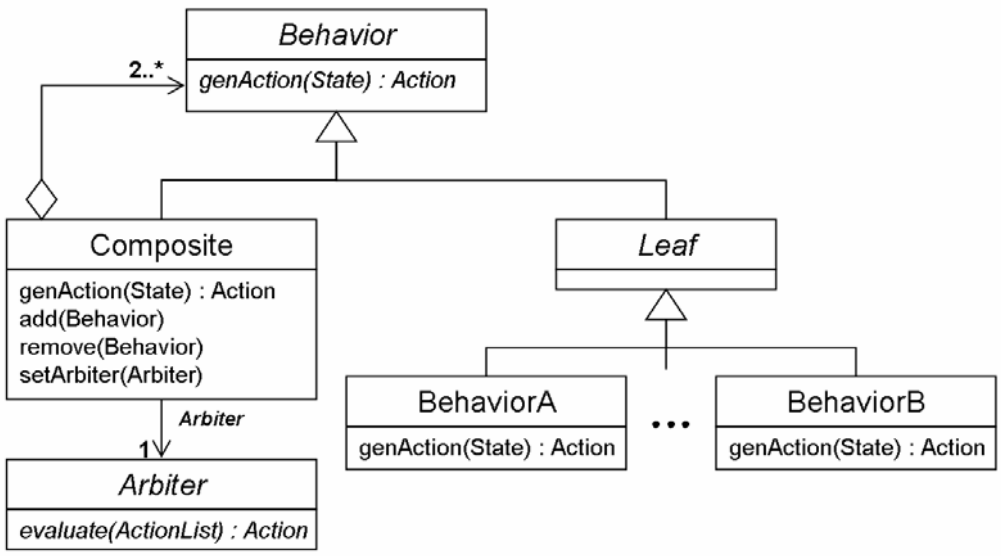



Figure 4.5: UML class diagram for the unified behavior framework.

The composite behavior acts as a joining element, allowing many behaviors to be formed into an arbitrated hierarchy. Its functional purpose is to maintain an ordered set of behaviors, $\boldsymbol{B} = \{b_1, .. , b_n\}$, and when asked to generate an action recommendation, it builds a corresponding set of proposed actions, $\boldsymbol{A} = \{a_1, .. , a_n\}$, which are collected by invoking the *genAction* method for each member of $\boldsymbol{B}$. Since the *Composite* class extends the abstract *Behavior* class, each returns a single *Action* object. Thus, an arbitration unit is used to determine a single action, $\boldsymbol{a'}$, from the set $\boldsymbol{A}$. The composite behavior then returns $\boldsymbol{a'}$ as its action recommendation. Because many arbitration techniques exist, the UBF does not attempt to embed a fixed approach into the composite behavior, rather it encapsulates the arbitration task as an associated arbiter module. This separation of the arbiter from the composite behavior allows the arbitration technique to be changed at any time by invoking the composite behavior's *setArbiter* method. In addition to *setArbiter*, the *Composite* class has other helper methods to manage the structure of a composite behavior.

Outwardly, the responsibility of an arbiter is to accept the set of actions, $\boldsymbol{A}$, via the evaluate method and return a single action recommendation, $\boldsymbol{a'}$. Individual arbitration algorithms are established by extending the abstract *Arbiter* class. This approach classifies all arbiters as belonging to the same family of arbitration algorithms, making them interchangeable. Internally, an arbiter can use each action's vote value and scale it

using an associated weighting to generate the ultimate recommendation, $\boldsymbol{a'}$. The set of weights, $\boldsymbol{W = \{w_1, .. , w_n\}}$, is a normalized set used to scale the affects of a behavior in relation to the other behaviors in the set $\boldsymbol{B}$. As an example, a vector summation arbiter and a utility based arbiter are presented below:

*Vector Summation*—a vector summation arbiter scales all the motor command recommendations made by the elements in $\boldsymbol{A}$ by the related weights of $\boldsymbol{W}$. The resulting values are summed to form the final action recommendation $\boldsymbol{a'}$. This approach allows values of $\boldsymbol{W}$ to be used as a means of tuning the contributions of individual behaviors to achieve the desired global behavior attributes.

*Utility Arbitration*—a utility based arbiter is considered winner-take-all selection approach, establishing $\boldsymbol{a'}$ by selecting the action recommendation with the greatest utility—where utility is calculated as the product of an action's vote value scaled by an associated weighting.

## *4.4 State and Action Interfaces*

A significant objective of the UBF is to create behaviors that are reusable in broader domains. The first step to towards reusability is to decouple behavioral logic from specific hardware. Like the proxies in Player/Stage [27], the State and Action classes provide behaviors with a generic interface for accessing sensor information and enacting motor commands. The abstract behavior interface in Figure 4.2 is structured in the spirit of reactive behaviors, accepting sensor input via the State and returning an action recommendation. Unlike the Player/Stage proxies that allow programs to query sensor values and submit motor commands, the State is strictly a shared information space while Actions are used to communicate a behavior's recommended motor commands back up the hierarchy without acting directly on the robot.

### *4.4.1 The State Interface*

The State is intended to be a shared information space similar to the Local Perceptual Space (LPS) used in Saphira [34]. The state is a multifaceted representation of the current environment, including: collections of decoupled sensor data, a fused sensor picture, positional information, goals, and so forth. Because the structure of the UBF

allows many behaviors to make action recommendations by independently evaluating the current state, behaviors are restricted from making changes to the state. The reason for this is that after proposing an action, behaviors never know if their recommendations are enacted, changed or ignored at higher levels of the hierarchy.

As a central component of the system, the State is expected to be in high demand, and thus it must remain a thread-safe [9] data repository that is free from logical calculations, using getter/setter methods to provide efficient access. While individual behaviors typically find a small number of state attributes relevant, the variety of attributes required by the population of behaviors is large. Additionally, the shared state quickly becomes a monolithic perceptual space when additional requirements are added to support routines for maintaining the state or performing activities like goal planning, mapping, navigation, etc.

### *4.4.2 The Action Interface*

The action class is a motor command interface that allows behaviors to propose sets of motor commands and indicate their current level of confidence. Contention of the motor commands is avoided by withholding access to the robot until the active behavior returns a single action to the controller. This approach requires that behaviors evaluate the shared State and propose motor commands via an intermediate action object. An action object consists of three distinct pieces: the set of motor commands, the vote field, and the platform specific motor control interface.

The main portion of the action interface is the set of motor commands. As an example, an action object is introduced in Figure 4.6 that supports commands for velocity and turn rate. In general, the set of motor commands for a particular domain supports motor commands for each degree of freedom. The external interface consists of getters and setters for each motor command and an additional getter which signals if a motor command has been specified. Out of bound values were initially used to indicate an unset motor command, however, the addition of an internal set/unset flag is less ambiguous. When a setter method is invoked, the associated *set/unset* flag is changed to *set*. This capability is useful for arbiters that blend actions.

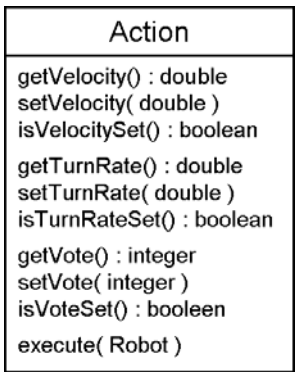



Figure 4.6: The public interface of an Action class that supports velocity and turn rate.

In addition to the set of motor commands, the action object carries a *vote* field which is set to indicate a behavior's intention to abstain or contribute to the system. The vote value can represent either a gradient value to indicate levels of activation or units of utility and as a unary value to indicate a binary yes/no vote.

The platform specific motor control interface is embedded within the execution method to keep behaviors from needing to know how to use the robot that they are being used to control. While behaviors can build action recommendations, they cannot unilaterally enact those recommendations because they do not hold a reference to the robot. This approach avoids contention by ensuring that the recommended set of motor commands returned by the controller's active behavior is the only one enacted on the robot. To do this, the controller invokes the execute method, giving the action object the authority to act on the robot directly. The action's execute method simply applies the current set of motor commands. The elements in the set of commands marked as *set* are enacted while *unset* elements are ignored. An example of an execute method that enacts velocity and turn rate commands is given by the following pseudocode:

```
execute (robot) {
  if (isVelocitySet) then robot.setSpeed(velocity);
  if (isTurnRateSet) then robot.setTurn(turnRate);
}
```

## *4.5 Building Behavior Structures*

To illustrate the mechanisms provided for constructing behaviors, this section demonstrates how the UBF is used to represent the typical behavior architectures: Subsumption, Motor Schema, Circuit Architecture, Action-Selection, Colony

Architecture and Utility Fusion. Each sub-section provides a short summary of the architecture and demonstrates an equivalent implementation in the context of the UBF. The goal is to demonstrate that each of the architectures can achieve a common interface, which allows autonomous robot system developers the flexibility to adapt to the current environmental conditions by selecting and using the most appropriate behavior structure.

### *4.5.1 Subsumption*

The Subsumption architecture [15, 17], described in section 2.2.1, advocates for a layered control system based on task decomposition rather than function, a radically different structure from the sense-plan-act approach. Figure 2.1 highlights this quintessential paradigm shift. Further, the parallel organization naturally promotes concurrent and asynchronous responses to sensor input, where each individual layer works to achieve its particular goal. Coordination between layers is achieved when complex actions (or higher layers) subsume simpler behaviors, or the low-level behaviors inhibit the higher layers. Fundamentally, Subsumption can be viewed as a competitive architecture using rule-based encodings and priority-based arbitration based on hierarchical priority [4].

To establish the Subsumption architecture in the context of the UBF, two fundamental intents of Subsumption need to be captured and preserved. The first major concept of Subsumption is the use of rule-based behaviors (or layers), and the second is the suppression/inhibition mechanism which allows layers to subsume or inhibit the outputs of other layers.

The translation is depicted by Figure 4.7 where the traditional Subsumption architecture is shown in (A) and a corresponding implementation under the UBF is shown in (B). Each task layer is represented directly in the UBF as an individual leaf behavior. A behavior builds an action object with the motor commands that it would execute if given control and when environmental conditions are suitable, it sets the vote field to indicate a desire to be considered for selection. The original suppression mechanism used for coordination exists as a simple priority arbiter associated with a composite node that groups the leaf behaviors together. By using the priority arbiter as the merging mechanism, a composite behavior that models Subsumption is formed,

where the action recommendation of the highest priority behavior that did not abstain is selected and returned by the composite node.

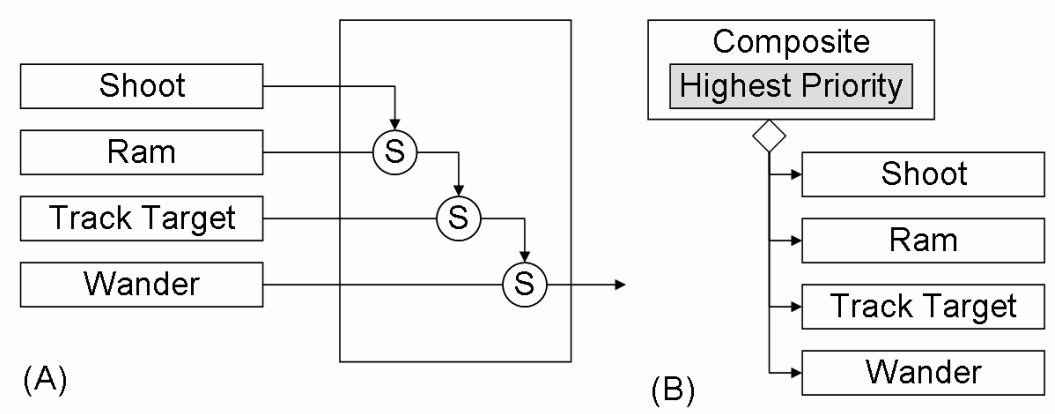



Figure 4.7: Equivalent implementations of Subsumption (A) Behavioral layers arbitrated via a suppression network; (B) A behavior hierarchy using a priority arbiter.

The use of a hierarchical structure to encapsulate each behavior layer preserves Brooks' original intention to be able to implement, test, and debug each layer independently. Additionally, because the behaviors in the hierarchy are independent, they are well suited for concurrent and asynchronous execution.

In the original Subsumption design, communication is allowed between layers, allowing feedback loops to exist. This feedback mechanism has received criticism because upper layers interfere with lower ones, which keeps each layer's design from being independent, and prevents the ability to test and debug layers independently [4]. Since the independence of behavioral layers is fundamental to the concept of Subsumption and the structure of the UBF, feedback loops are not represented.

### *4.5.2 Motor Schema*

The motor schema architecture [2], described in section 2.2.2, is a cooperative control approach which allows for the simultaneous pursuit of multiple goals. Under this architecture, behavioral primitives are captured as vector fields that support specific perception tasks (e.g. obstacle avoid, move-to-goal, stay-on-path, etc.) which are arbitrated as a vector summation and normalization of a continuous potential field. This approach is useful in navigation tasks where a path to a goal must be discovered and obstacles exit on the direct path to the goal.

To enact the motor schema architecture in the context of the UBF the two fundamental aspects need to be captured and preserved. The first is the motor schema architecture's use of perception schemas to capture the governing motion effects around goals, obstacles, wall, etc. and the second is the use of vector summation and normalization as a means of coordinating motor commands.

The transformation is depicted in Figure 4.8 where the traditional motor schema architecture is shown as (A) and the corresponding implementation under the UBF is shown as (B). Since perception schemas are already modular and independent of one another, each one is represented as a leaf behavior that returns its action recommendation. Because the resultant vector field is normalized, each schema can set the vote field of its action if an event that attracts or repels the robot is detected, otherwise it abstains. The original summation and normalization mechanism used for coordination is captured directly as an arbiter that generates a fused command response. By grouping the schemas together under a composite node that uses a command fusion arbiter the motor schema architecture is effectively implemented within the context of the UBF.

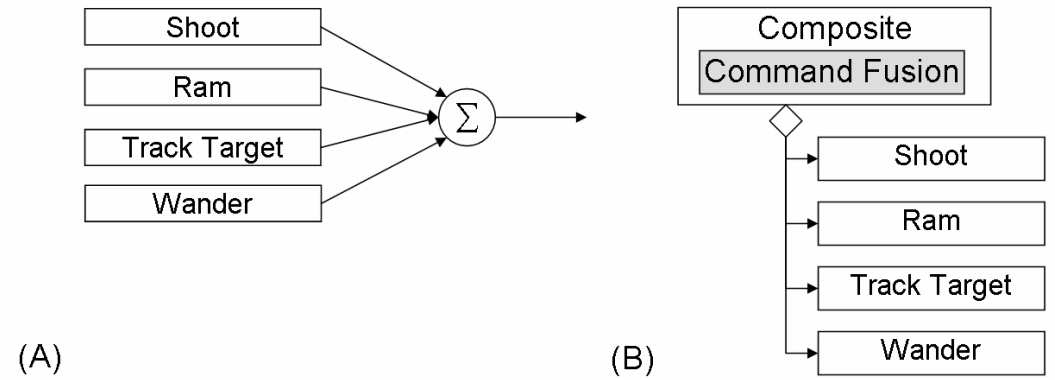



Figure 4.8: Equivalent implementations Motor Schema (A) Independent motor schemas coordinated via summation and normalization; (B) A behavior hierarchy using a command fusion arbiter.

In the original implementation of the motor schema architecture, there are mechanisms that allow the effect of individual schemas to be weighted as a means of tuning the global behavior. When implemented under the UBF, this is achievable by weighting the affects of the associated behaviors within the arbiter. Consider a simple schema pair where goals attract and obstacles repel. By increasing the weights associated

with the obstacle-avoid schema, the robot will swing wide of obstructions and may navigate around cluster of obstacles. In contrast, by increasing the weights associated with the goal-seeking schema, the robot may approach obstacles more closely or try to weave between groups of small obstacles. By making adjustments to the weights within an arbiter, the individual schema remains generic and reusable. This is significant because it allows distinct branches of control to use the same schema with different affects in each branch.

Additionally, this implementation maintains the key strengths of Arkin's original motor schema architecture. The use of modular perception schemas supports parallel and distributed computation, runtime flexibility and delivers reusable components that are stored and called from behavior libraries [4]. The motor schema representation described above maintains all of these qualities.

### *4.5.3 Circuit Architecture*

The circuit architecture [33], described in section 2.2.3, is an abstraction approach that groups reactive behaviors and logical formalisms into arbitrated collections. Because arbitration occurs at each level of abstraction, developers can build hybridized bundles of unlike approaches into mediated hierarchies that are combinations of reactive approaches, logical formalisms, and situated automata [4].

To enact the circuit architecture in the context of the UBF, two fundamental aspects need to be captured and preserved. The first is the ability to create bundles of either reactive behaviors or logical formalisms, and the second is its use of hierarchical mediation. The key components of this architecture translate directly into an implementation using the UBF, since leaf behaviors can be implemented as either a reactive behavior or as a logical formalism. Additionally, composite behaviors are mediated hierarchies since each has an associate arbiter. The ability for each composite behavior junction to use a different arbitration technique allows the leaf implementations to be arranged into synonymous hierarchical structures.

The designers original motivations were to allow for modularity and incremental development, responsiveness via tight coupling of sensing to action, and robust designs that could perform despite unexpected environments or hardware failure. Due to the

similar design structure and the low overhead of using the UBF components, the original design motivations are preserved.

### *4.5.4 Action-Selection*

Action-selection [37], described in section 2.2.4, uses activation levels as a dynamic mechanism of competitive behavior selection. Individual behaviors are grouped as competence modules that respond when predefined conditions are detected. These requirements can be simple environmental triggers, sequential observations, the existence of higher level goals, previous or potential success, etc. When a module's preconditions are satisfied, an associated action sequence is initiated at a given activation level. Activation values can be instantaneously reported while the trigger condition is present, can persist for a set period of time or can have various decay rates. Action coordination is achieved by selecting the competence module that currently has the highest activation level. Because there is no predefined layering or order of execution when used in dynamic environments, this gives the global behavior a greater emergent quality. By basing priority on events in the environment a robot can suddenly and deliberately respond to unique conditions or changes in the environment.

To implement the action-selection architecture in the context of the UBF the two fundamental aspects are captured. The first is the organization of response rules into competence modules and the second is its use of activation levels to coordinate action selection.

The modularity and independence of individual competence modules allows them to be implemented directly as individual leaf behaviors. Action-selection's use of competing activation signals has no direct equivalent under the UBF. To establish an equivalent ability, the activation level is embedded in the action recommendation using the behavior's vote field to indicate its current activation level. The arbitration scheme used by action-selection is represented in the UBF as a highest activation arbiter associated with the composite behavior that groups related competence modules together. Being a winner-take-all approach, the arbiter evaluates the vote value for each action recommendation and returns the action with the highest value. This approach lends itself to a hierarchical structure of competence modules that can be several layers deep.

### *4.5.5 Colony Architecture*

The Colony architecture [20], described in section 2.2.5, is a direct descendant of Subsumption, allowing higher layers to suppress lower layers but eliminates the ability of lower layers to inhibit higher ones. As a result of enacting the suppression only approach, the colony architecture breaks away from the total ordering of layers found in Subsumption and permits hierarchical arrangements of behavioral priorities [4].

This architecture is naturally represented in the context of the UBF, specifically the priority based hierarchies. By capturing the logical control layers as leaf behaviors, functional branches of control are then formed via composite nodes using highest priority arbitration. Figure 4.9 depicts a Colony architecture design, grouping the shooting and target tracking behaviors as a separate control structure. A traditional suppression network is shown in (A) and an equivalent representation using the UBF structure is shown in (B).

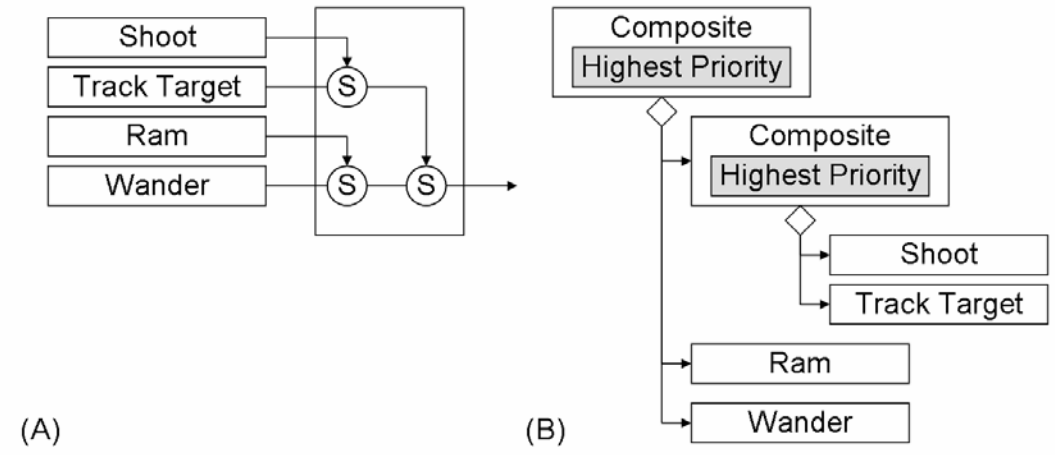



Figure 4.9: Equivalent implementations of the Colony Architecture (A) Priority based hierarchy via a suppression network; (B) A highest priority control structure via the UBF.

### *4.5.6 Utility Fusion*

The utility fusion architecture [44], described in section 2.2.6, is an expansion of the DAMN architecture [43] which distributes action selection via utility instead of priority-based or command-fusion arbitration approaches. Under utility fusion an arbiter collects utility votes for proposed actions from the associated evaluation behaviors. Each behavior uses its own criteria to independently assess the utility of a future state that results from taking a given action. The action that collects the highest overall utility is

enacted by the arbiter. This approach provides the arbiter with much richer evaluation information because actions are selected by how they best satisfy the system's overall goals, or how the action best meets the goals of all of the behaviors. For example, if several behaviors assign a modest utility to one action while only one behavior assigns a high utility value to a second action, the utility based arbiter will select the action which accumulated the largest total utility. In most cases the first action will be selected because the overall utility from several moderate votes is greater than a high utility vote from a single behavior. This is an appropriate assessment because the first action simultaneously meets more of the system's overall goals.

Under the utility fusion architecture, Rosenblatt intends that behaviors evaluate candidate future states so that they do not need to know the system kinematics or the specific motor commands to achieve the proposed state [44]. This modular approach binds the abilities of behaviors to the detection of favorable/unfavorable conditions within the environment and divorces them from implementation details. The benefit of this approach is that behaviors become modular and reusable across systems that employ the utility fusion architecture.

To capture the DAMN and utility fusion architectures under the UBF, the original aspects and intent need to be captured and preserved. This is a difficult problem because sub-behaviors are being asked to evaluate projected states rather than the canonical shared perceptual space. While the evaluation behaviors can be represented directly as leaf behaviors that indicate their utility assessment using the vote field embedded in the action object returned by the behavior (any proposed action values are ignored), some mechanism that understands the kinematics of the robot must be implemented such that it can generate predictive state representations for the leaf behaviors to assess. This approach is quite different than the other architectures because the behaviors are evaluating future states and not the canonical shared perceptual space.

This central predictive mechanism would hold a discrete set of actions and the ability to generate predictive state representations for each of those actions. The predictive states are then given as input to the evaluation behaviors. Consider an architecture that supports six actions (strong-left, weak-left, straight, weak-right, strong-

right, and stop) and uses five evaluation behaviors to assess the utility of a proposed action. The predictive mechanism must generate the six future states that would result from the execution of each action. The utility of each future state is assessed as a sum of the utilities returned by the five evaluation behaviors and is embedded into its associated action using the vote field. The enacted action is selected using a simple highest-utility arbiter that evaluates the vote values for each element in the action set.

The unique structure of this predictive element gives unprecedented responsibility to the joining component; it must understand the kinematics of the implemented system and be able to model the consequences of its actions on the environment. The best representation of this element in the UBF results in extending the existing composite behavior and expanding the *genAction* method to perform the predictive modeling and utility assessments for each member of the action set.

Rosenblatt suggests that this predictive approach can be expanded to make utility assessments for chains of action steps. While the implementation approach above also expands to support this predictive capability, the effects of additional deliberative computation must be considered in order to avoid adversely affecting the responsiveness of the robot.

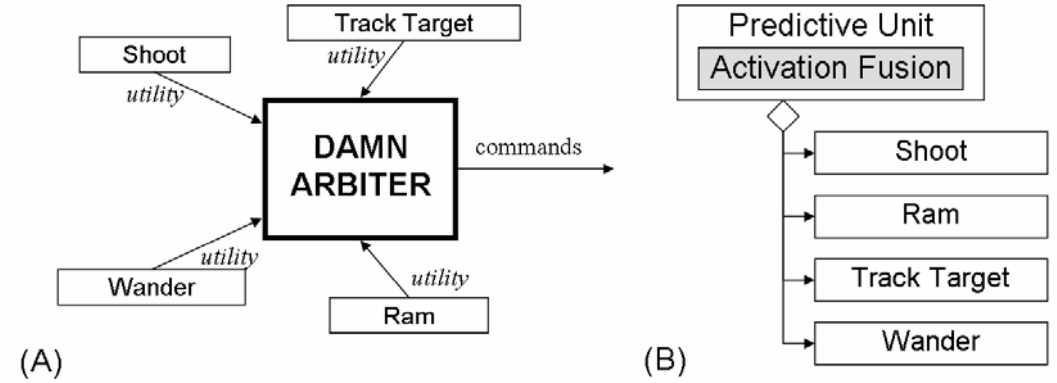



Figure 4.10: Equivalent implementations of Utility Fusion (A) Evaluation behaviors assess utility; (B) Behavior hierarchy using a predictive unit and a highest-utility arbiter.

## *4.6 Sequential Implementation*

The concept of the UBF is to guide the development of a collection of primitive behaviors that can be used interchangeably and/or be composed with other behaviors in the collection to form complex behaviors. Up to this point, only the interaction of the

classes in the framework has been discussed because the UBF is intended to ensure the modularity of behaviors and not to specify their implementation. Developers, however, do need to consider the details of implementation to ensure that the performance needs of the system are being met.

This section presents the most direct method for implementing behaviors in the UBF. The use of sequential programming, which embeds all of the behavioral logic within the *genAction* method, makes the flow of execution though the UBF easy to understand. An alternative implementation for concurrent and real-time systems is discussed in the next section. Because the focus of the UBF is not to specify implementation, developers are not bound to one approach. A heterogeneous mix of these approaches can be used to compose a behavior structure as long as it also meet the system's performance requirements.

The concept of the sequential approach is to evaluate the current state and generate an action recommendation when the *genAction* method is invoked, referred to as *on invocation*. This is the simplest and most direct approach for implementing a primitive behavior and is effective for simple or light-weight algorithms in domains with discrete time steps. The following pseudocode provides an example of its implementation structure:

```
genAction(state) returns Action {
  // Evaluation Operation...
  return action;
}
```

The on invocation approach presents an immediate drawback because there is no safe guard for detecting and avoiding repetitive work. When considered in the context of the UBF, it is likely that a behavior will be used as a repetitive element within a behavior structure, causing it to evaluate the same state and generate the same action recommendation on each invocation.

To alleviate the burden of repetitive computations the last action recommendation and the associated time stamp of the state are kept by the behavior. Within the *genAction* method a conditional statement is added to check the state's current time stamp. If the check indicates that this state has already been evaluated, then the stored action is

returned, bypassing the evaluation algorithm. This approach is useful for reducing the computational requirements of a behavior structure in turn based environments where evaluations are made in discrete time steps. The pseudocode below expands on the previous implementation, adding the conditional check:

```
genAction(state) returns Action {
  if (state.getTime == lastTime) return action;
  else
  {
    lastTime <= state.getTime;
    // Evaluation Operation...
    //
    return action;
  }
}
```

The on invocation approach is useful because of its simplicity, making the implementation of stable behaviors easy to understand. However, this structure is best suited for turn-based domains that have discrete time steps. For continuous time domains, this approach can put a computational burden on the system because the entire hierarchy must be evaluated. The next section discusses a design for leaf behaviors that is usable for concurrent and real-time execution.

## *4.7 Concurrent and Real-Time Implementation*

As demonstrated by YARA [18], the ability of low-level control behaviors to reliably run at periodic intervals is crucial to the safety and reliability of robots operating in continuous domains that are both dynamic and unpredictable. The sequential implementation of the UBF presented above, while straightforward, necessarily ties the rate at which a controller can request an action recommendation to the computational time of the hierarchy.

This section presents an asynchronous implementation of the UBF intended to be employed within robot architectures that leverage concurrent and real-time scheduling. The fundamental change over the sequential approach is that the computational logic of the leaf behaviors is moved out of the *genAction* method and into a separate thread of execution that is run periodically via a scheduler. With each leaf behavior scheduled to evaluate the environment at appropriate periodic intervals, the current evaluation result

can be obtained repeatedly via the *genAction* method. This approach treats the set of base behaviors as a pool of independent worker threads that execute as concurrent and potentially simultaneous processes.

This change does not affect how a controller requests an action recommendation from its active behavior. In fact, this implementation makes the call to *genAction* quite fast because it need only traverse the behavior hierarchy to collect and arbitrate the current action recommendations down to a single recommendation. Additionally, this approach divorces the rate at which a controller requests action recommendations from the rate at which each leaf behavior evaluates the environment.

Using this implementation approach, the structure of a leaf behavior has two major parts: the *genAction* method and the run method. The *genAction* method is a requirement of the abstract behavior class and provides asynchronous access to the behavior's current action recommendation. The run method, called periodically by a scheduling process, evaluates the current environment and updates the current action recommendation.

The pseudocode below provides a structural example for a schedulable behavior:

```
genAction(state) returns Action {
  return action;
}

run(state) {
  // Evaluation Operation...
  //
  write.lock;          // Locks are required to protect
  action <= current;   // against interference caused by
  write.unlock;        // concurrent access of genAction.
}
```

An asynchronous implementation naturally raises the question, "How frequently should a controller poll its active behavior?" Unfortunately, there is no best answer, but all solutions must consider the level of uncertainty in the current environment. One approach is to request an action recommendation at twice the rate of the fastest periodic environment evaluation. This solution is based on the principle of the Nyquist sampling rate [42] and assumes that the periodic schedules of the base behaviors are adjusted at runtime to match the environment's current level of change. In rapidly changing environments, this approach allows low-level processes to execute at shorter periodic

intervals, increasing the computational time used by reactive control routines. In more stable environments, the scheduler can set low-level processes to execute less frequently, making computational time available to higher-level planning processes.

When used with real-time schedulers, this approach can potentially enter a condition known as *priority inversion*. Priority inversion is a problem that can emerge in real-time systems that have multiple processes, running at various priorities and are attempting to access shared data protected by a blocking semaphore. Priority inversion occurs when a low priority process is holding a lock that a higher priority process needs to progress. Real-time schedulers are designed to preempt the running processes when a higher priority process is ready to run. If the higher priority process never yields while waiting for a lock that is held by a preempted process, the lower priority process never has the chance to run and release the lock, creating a deadlock condition. The solution is to employ *priority inheritance*, a mechanism provided by many real-time operating systems to detect priority inversion. To bypass this deadlock situation, the operating system temporarily raises the priority of the lower priority process, so that it may run. Once the lock is released, the process reverts to its original priority.

This implementation of the UBF uses the independence of leaf behaviors to establish their evaluation processes as threads of execution that can be scheduled to execute at various periodic rates. Such an approach is well suited for applications that use a real-time operating system to cope in dynamic and continuous time environments. The periodic evaluation approach presented associates the computational requirements of the system with the established execution schedule. Regardless of the frequency that the controller requests an action recommendation and the number of times that the same leaf behavior is used within the active structure, evaluations are only performed once per period. The drawback to using this approach is the added complexity of using concurrent processes, which can be minimized by using established design approaches.

## *4.8 Summary*

The UBF is a structural guide that applies standard software engineering approaches to simplify development and testing of reactive behavior modules for autonomous robots. At the highest level, it uses a strategy pattern [25] to establish a

family of interchangeable behaviors. Additionally, the UBF addresses the need for scalability by providing construction tools that allow robust structures to be formed as arbitrated hierarchies of small, highly focused components. The use of the composite pattern [25] then ensures that the resulting structures are scaleable and belong to the established family of behaviors. This approach eases design complexity, allowing atomic behaviors to be designed, implemented and tested independently and then joined together to produce rich and coherent behaviors. The ease with which components can be formed into stable structures encourages reuse and experimentation.

By hiding the implementation details of individual behaviors behind a common interface, each behavior implementation is developed and tested independently, allowing independent, and possibly parallel, development teams the freedom to use the behavior system that they feel will best achieve their design goals. Additionally, since the scope of each behavior is focused, code complexity is reduced which in turn eases testing requirements. Once established, any compatible robot controller can use a behavior as an interchangeable behavior element.

The ability of the UBF to present behaviors as interchangeable elements inherently supports planning and allows a low-level controller to provide the sequencing processes a robust method for altering the controller's apparent immediate goal. By observing and evaluating the shared perceptual space, the sequencer can form the overall behavior of the robot by changing the active behavior process [23]. The idea of sequencing a series of simple tasks into a chain that yields a higher order goal is based on an approach originally presented by the Reactive Action Packages (RAPs) system. Unlike RAPs, the UBF does not require low level behaviors to report success/failure because reactive behaviors simply act within their current environment (either well or poorly) without knowledge of what constitutes success or failure in a given system. For this reason, the entity that has knowledge of the system's higher-order goals (i.e., the sequencer) is expected to monitor the progress of the active behavior within the environment to assess success or failure. Additionally, the sequencer can adjust the criteria of success/failure on the fly rather than relying on assessment logic embedded within a reactive behavior a priory.

In the next chapter, the capabilities of the UBF are demonstrated using three case studies. The first case study demonstrates how modular designs encourage code reuse, rapid prototyping, and experimentation. The second case study applies a genetic program to automate the discovery of effective behavior structures for given domain. The final case study demonstrates the ability to establish a safe and dependable basis of control by implementing a behavior-based controller as a set of real-time tasks that remain predictably responsive regardless of the system's computational load.

# V. Results

This chapter demonstrates the capabilities of the unified behavior framework (UBF) using three experiments intended highlight individual capabilities of the design. The first section demonstrates how arbiter selection affects global behavior. Section 5.2 demonstrates how a genetic program is employed for the automatic discovery of effective behavior structures. The application of the UBF within the context of a real-time operating system is demonstrated in section 5.3. The final section reiterates the results of the three experiments as they apply to the concepts of the UBF overall.

## *5.1 Case Study I: Arbiter and Structural Variation*

The burden of developing and testing behavior-based controllers is in the level of details required to implement a robust and coherent basis of control for a given robot in a given domain. Designs that blend the hardware interface, kinematics, and behavioral logic become monolithic controllers that are not reusable. The unified behavior framework (UBF) advocates that low-level controllers encapsulate their behavioral logic and provide generic sensor and motor interfaces to the system. This design allows controllers to dynamically reconfigure their behaviors without changing the design of the controller. The modular design of the UBF supports the development of highly focused behaviors and arbiters that are subsequently combined into structures that outwardly display a robust behavior attributes.

This experiment uses an adaptation of the Robocode robot battle environment [40] to demonstrate how the modular design of the UBF supports reuse and composition, allowing a variety of dynamic behavior structures to be constructed and evaluated from a set of basic behavior/arbiter elements. This experiment also enacts representations of well known behavior architectures in such a way that they conform to the UBF's abstract behavior interface, which makes each one an interchangeable member in a family of behaviors. As a byproduct of this demonstration, the importance of selecting an effective arbiter is highlighted by the radical changes in the outward attributes of the behavior structures as different arbiters are employed for the same behavior structure.

The discussion of this experiment is broken into four sub-sections: an explanation of the Robocode modification to allow velocity based motor control, a listing of the behavior/arbiter components available for this experiment, a description of the behavior structures formed from these components, and the observations of the outward behavior for each structure along with an assessment of the performance gap that emerges.

### *5.1.1 Robocode Adaptation*

Robocode was chosen as a simulation environment because it provides a dynamic, environment in which different control architectures are compared by allowing them to interact dynamically in battle, see the screen capture in Figure 5.1. However, the current application is not useful for experimenting with established robot control architectures, because rather than accepting motor commands, commands are discrete requests that set a robot to turn left 90 degrees, or travel a set distance and then stop. This motor interface is atypical of standard robot motor control mechanisms. For this reason, the motor interface of Robocode version 1.0.7 was adapted to allow for a velocity-based approach, it now accepts commands that specify the desired velocity and rate of turn for the chassis as well as the turn rate for the gun turret and the radar. Once set, these rate based values persist until changed.

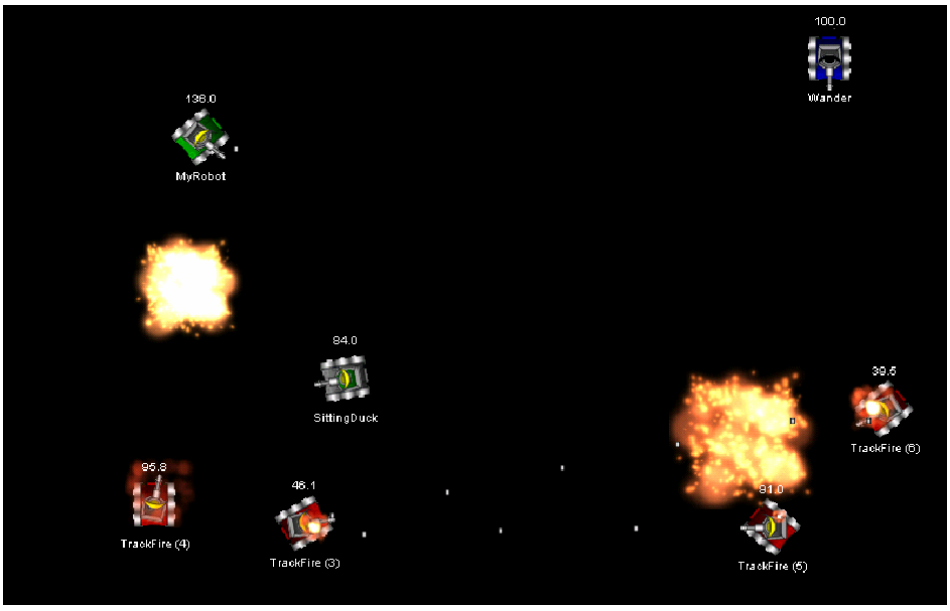



Figure 5.1: Screen capture from Robocode of a ten-on-ten robot melee.

The physical limits of acceleration that restrict a robot from instantaneously achieving a new velocity or turn rate are maintained by the new motor interface. For example, if a robot is moving in reverse with a velocity of –2 it cannot instantaneously change its velocity to +8. A command that sets the robot's velocity to +8 is accepted, but the change is not instantaneously reflected in the robot's motion. This concept is the same for the chassis turn rate, the gun turret turn rate, and the radar turn rate: each has unique acceleration parameters and maximum rate values.

### *5.1.2 Description of Elemental Components*

Using the UBF interface, five elemental behaviors and six arbitration techniques are developed and tested as independent components. The functionality of each component is described below and then used to build specific architectures for the comparison in Section 5.1.4. The behaviors are:

*Ramming*—when another robot (with a lower energy level) is detected, this behavior causes our robot to turn towards the other and charge towards it, attempting to cause damage by hitting it.

*Shoot*—causes our robot to fire on another when the target is less than three degrees off bore site and is within range. The power committed to the bullet is reduced as a function of the target off bore site angle and as a special case, when the target is close the shot is taken at max power regardless of the angle error.

*Scan*—causes the radar to oscillate in a twenty degree arc around the bore site of the gun. As the gun rotates the active radar scan area tracks with it.

*Target Tracking*—has two operating modes. When no target is detected, the default mode turns the turret in a clockwise direction. When a target is detected, the target tracking behavior causes the gun turret rotation to slow or reverse its direction in an attempt to continue tracking the target. This behavior sets the turret turn rate to be one-third of the current off bore site angle, as a target's own motion causes the off bore site angle to increase, the turret turn rate increases accordingly in an attempt to track the target.

*Wander*—is always active, attempting to performs a series of "S" turns across the battlefield. When a wall is encountered, the polarity of the velocity is flipped.

The length of the arc is randomly selected to be between 30 and 120 degrees before changing the direction of turn.

The arbitration techniques developed are:

*Monte Carlo*—is a stochastic arbitration technique that uses fitness proportional random selection to activate one sub-behavior for a period of time. At the end of the period another random selection occurs, activating the chosen sub-behavior for the current period.

*Highest Priority*—is a winner-take-all arbiter that returns the action set of the highest priority behavior indicating a desire to act, regardless of vote value. The recommendations of lower priority behaviors only execute if higher priority behaviors abstain.

*Priority Fusion*—is a semi-cooperative arbiter that uses priority based arbitration on a per motor command basis. Unlike the highest priority arbiter above, priority fusion builds a new action set that allows the unspecified action fields of higher priority behaviors to be filled by lower priority action requests.

*Command Fusion*—is a cooperative arbitration approach that uses summation and normalization of proposed motor commands to derive the resultant set of motor commands. The input of all contributing behaviors are used on a per motor command basis to form the resultant command vector.

*Highest Activation*— is a winner-take-all arbiter that returns the action set with the highest vote value. This approach provides a dynamic mechanism for competitive selection by allowing behaviors to indicate their urgency for activation. Associated behavior weights are used to internally tune global performance by scaling the votes of behaviors that either over or under vote. The concept of activation levels is synonymous with the concept of utility in market based systems.

*Activation Fusion*—is a semi-cooperative arbiter that uses a highest activation selection approach on a per motor command basis. Unlike highest activation, activation fusion builds a new action set, allowing the motor commands left unspecified by the behavior with highest level of activation to be set using the

recommendations of behaviors with lower activation levels. When used with market based systems, this technique is easily referred to as utility fusion, but risks confusion with Rosenblatt's utility fusion [44] behavior architecture.

### *5.1.3 Behavior Structures*

The ability of the UBF to easily assemble elemental components into operationally sound structures encourages developers to experiment with different structural arrangements. This approach isolates the task of trouble shooting a system's outward behavior to the structural arrangement of independent control elements that have been previously validated. The experiment in this section primarily demonstrates how structural changes affect the global attributes of a behavior. Additionally, it demonstrates the profound affect that structure and arbitration have on the overall effectiveness of base behaviors.

To make a quantitative comparison of the relative effectiveness of one behavior structure to another, the experimental structures are evaluated using their performance in relation to a benchmark that was crafted by an expert to operate coherently within the domain. The benchmark's control architecture is shown in Figure 5.2a, and consists of the wander behavior, the ramming behavior, and the track-fire behavior arbitrated by an activation fusion arbiter. The observed behavior of the benchmark has two operating modes, one that wanders the battlefield attempting to track and shoot opponents and another that aggressively charges towards a weaker opponent with guns blazing.

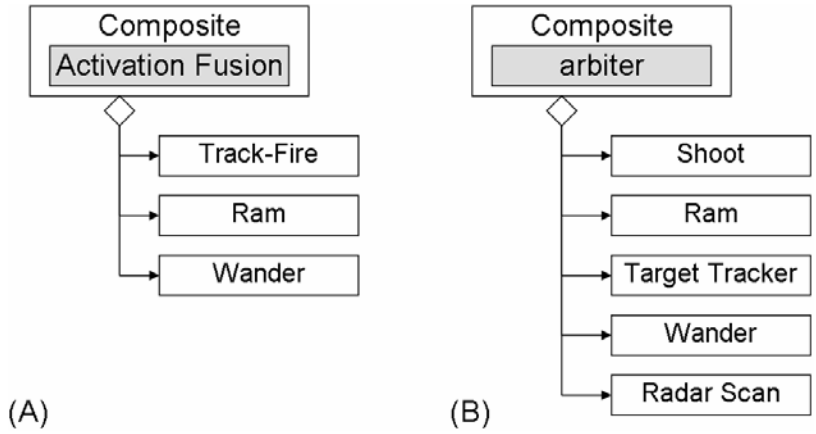



Figure 5.2: (A) The benchmark's behavior structure; (B) The standard behavior structure, each of the six arbitration approaches is evaluated in turn by setting the arbiter field.

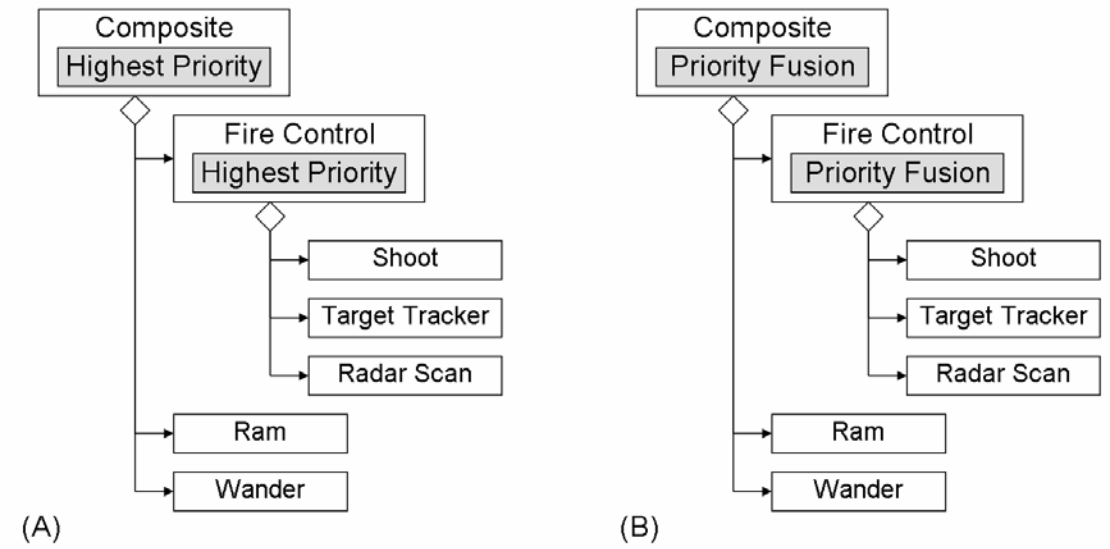



Figure 5.3: Two representations of a traditional Colony architecture (A) Colony A is a hierarchal structure using highest priority arbitration; (B) Colony B is an adaptation of the traditional hierarchy that uses priority fusion arbiters to support the activation of the sub-behavior requests on a per motor command basis.

Using the standard behavior structure in Figure 5.2b, each of the elemental arbiters can be applied at the composite node to form six distinct behavior structures that each have shoot, ram, target tracker, wander and radar scan as their base behaviors. Each structure is named for the arbiter in use. In addition to these six standard structures, the priority-based hierarchy approach specified by the colony architecture [20], is used to demonstrate how hierarchical control structures can be formed using the UBF. Two variations of the colony architecture are introduced, each grouping the three shooting related behavior elements into a fire control branch. The first structure (called Colony A) uses strict priority based arbitration at all levels of the hierarchy and is shown in Figure 5.3a. The second structure (called Colony B) is identical but uses priority fusion arbitration instead, see Figure 5.3b, and shows that the UBF can extend beyond the discussed behavior-based architectures.

### *5.1.4 Results*

The quantitative measures for each structure's performance, measured relative to the capabilities of the benchmark behavior, are presented by Table 5.1. The rating of each member is measured using a series of 250 battles against a robot running the benchmark

behavior control architecture. Rankings are the percent difference of the benchmark's score; values above zero indicate superior combat skills while below zero ratings indicate an inferior level of performance. A clear performance gap can be seen in Figure 5.4 where the four fusion-based structures are stratified above the four competitive structures. Subjective observations about a behavior's apparent coherence are also made to aid in the discussion of the quantitative results.

Table 5.1: Performance of behavior structures relative to the benchmark.

| **Behavior Structure** | **Architecture Representation** | **Rating** (avg ± stdev) | | **Action Selection** |
|---|---|---|---|---|
| Colony B | – | 104% | ±12.5% | Fusion-Based |
| Priority Fusion | Circuit Arch | 104% | ±13.1% | Fusion-Based |
| Activation Fusion | Utility Fusion | 104% | ±13.6% | Fusion-Based |
| Command Fusion | Motor Schema | 99% | ±14.6% | Fusion-Based |
| Benchmark | – | 0% | ±17.6% | Fusion-Based |
| Highest Activation | Action-Selection | –19% | ±28.5% | Competitive |
| Highest Priority | Subsumption | –19% | ±29.1% | Competitive |
| Monte Carlo | – | –27% | ±26.7% | Competitive |
| Colony A | Colony Arch | –47% | ±30.9% | Competitive |

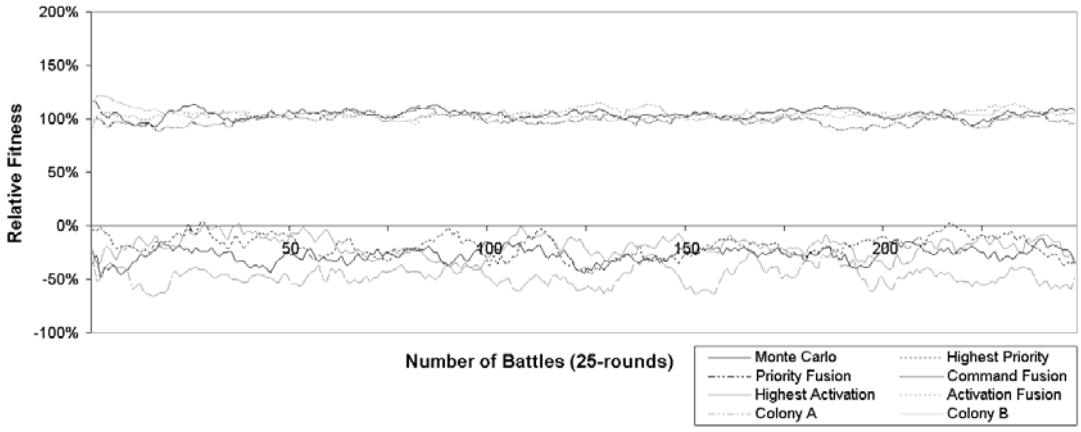



Figure 5.4: Performance of a standard behavior structure using various arbiters.

*5.1.4.1 Monte Carlo Structure*

The member using the Monte Carlo arbiter randomly gives control to a single sub-behavior branch for set intervals before randomly selecting another. This structure performs poorly against the benchmark, achieving a rating of –27% ±26.7%. Outwardly, this behavior structure delivers an incoherent behavior, switching between its elemental behaviors. The reason is that several of the base behaviors are intended to be used in

combination with each other. The winner-take-all nature of the Monte Carlo arbiter does not allow it to switch between complementary behaviors at critical moments, and thus does not achieve the larger effect that is intended. For example, the member is able to track a target but begins to wander without shooting at the opponent. During the period when the shoot behavior is active, the robot fires when an opponent crosses its path but is unable to track the target and continue the attack. This arbitration approach is suggested for use when the individual sub-behaviors are robust enough to be used as standalone behaviors.

#### *5.1.4.2 Highest Priority Structure*

The member using the highest priority arbiter establishes a layered Subsumption architecture, prioritizing its sub-behaviors from highest to lowest: shoot, ram, tracker, scan then wander. This structure performs poorly against the benchmark, achieving a rating of –19% ±29.1%. The outward attributes of the member show its ability to track and fire on a target by interleaving priority shoot commands with tracking commands. Its weakness is that the third highest priority behavior (tracker) is always active and starves its two lowest (wander and scan). Additionally, the winner-take-all approach keeps specialized behaviors like ram from capitalizing on the tracking efforts being done by other behaviors. In fact, ram effectively disrupts the target tracking task by suppressing it to move towards the opponent but does not continue the tracking effort, subsequently loosing the target and disrupting its own ability to act. Like Monte Carlo this arbitration mechanism is better suited for organizing task specific behaviors that are coherent as standalone control structures.

#### *5.1.4.3 Priority Fusion Structure*

The member using the priority fusion arbiter, still uses the suppression of lower priority requests for higher ones, but on a per motor command basis. This difference effectively establishes separate circuits of control for each motor command, allowing lower priority behaviors to remain active unless overridden by a higher priority process. This structure performs well against the benchmark, achieving a rating of 104% ±12.5%. The outward attributes of the member appear much more robust and natural than the

strictly competitive constructs like highest priority. The member regularly switches between a charge and a wander roll without disturbing its ability to track and fire on an opponent. The rating jump is attributable to the ability of the arbiter to allow independent layers to be concurrently active, allowing the global behavior to avoid the starvation condition that cripples the highest activation structure.

#### *5.1.4.4 Command Fusion Structure*

The member using the command fusion arbiter, an approach modeled on the motor schema [2] architecture, generates a global action recommendation that is a normalized summation of all contributing sub-behaviors. Empirically, this structure performs well against the benchmark, achieving a rating of 99% ±14.6. The outward attributes of the member are robust and coordinated, allowing combinations of target tracking, shooting and movement routines to occur simultaneously. The primary weakness of this design is in the resultant motion control, which appears hesitant and shaky. This attribute of its behavior is attributable to the conflicting directives that cancel or dampen each other, making it a vulnerable target for a time.

#### *5.1.4.5 Highest Activation Structure*

The member using the highest activation arbiter, an approach modeled on the action-selection [37] architecture, implements a winner-takes-all approach that bases selection on weighted activation signals. This structure performs poorly against the benchmark, achieving a rating of –19% ±28.5%. The outward attributes of this member show that it is able to interleave tracking with shooting, but that it starves the scan and wander functions because the target tracking behavior maintains a higher activation level at all times. This design suffers further from an inability to produce coherent motion control. Like the highest priority member, the ramming behavior is enacted when a weaker opponent is detected, but lacks of ability to continue its tracking effort and looses target to be lost. In addition to thwarting the tracker, the failed charge action causes the robot to drive in pointless circles because wander is unable to contribute coherent motion control due to starvation.

#### *5.1.4.6 Activation Fusion Structure*

The member using the activation fusion arbiter is essentially using the standard highest activation approach, but is applying it on a per motor command basis. This allows the motor commands left unspecified by the highest voting behavior to be filled using the recommendation of behaviors with lower activation levels. This structure performs exceptionally well against the benchmark, achieving a rating of 104% ±13.6%. The outward attributes of this member show a robust blending of independent behaviors acting at critical moments to produce effective emergent behaviors that are not expressly provided in code. As an example, this structure effectively couples the ability of the ramming function to approach the opponent with the shooting and tracking behaviors effectiveness at close range. Alternatively, when the member is weaker than the opponent, it continues to track the opponent, but is less aggressive, wandering at a safe distance, seemingly waiting to regain the advantage.

#### *5.1.4.7 Colony Structure A*

This member uses the hierarchical behavior structure shown in Figure 5.3a to represent the traditional colony architecture design. It groups the behaviors related to shooting into a fire control structure that is alongside the ram and wander movement behaviors. This structural arrangement performs poorly against the benchmark, achieving a rating of –47% ±30.9%. The outward attributes of this member show that its fire control substructure is actively tracking and firing on the opponent from a stationary position. Its stationary nature is due to the starvation of the motion behaviors. Since the fire control substructure is given the highest priority it must yield, which it never does because it contains the always-active target tracker behavior, to allow the motion behaviors to execute. Within the fire control structure, the scan behavior experiences starvation as well, effectively basing this member's behavior on two of its five sub-behaviors. The stationary attribute of this member is the largest reason for its poor ranking in relation to the benchmark.

#### *5.1.4.8 Colony Structure B*

This member is an adaptation of the colony architecture that uses priority fusion arbitration to alleviate the affect of starvation caused by the winner-take-all approaches.

As with the Colony architecture A, it groups the shooting behaviors together as a high priority fire control structure, but does not starve the motion control elements because the resulting action returned by the root of the structure represents the highest priority requests for each motor command. This structural performs exceptionally well against the benchmark, achieving a rating of 104% ±12.5%. The empirical ranking and the outward attributes of this member closely resemble the activation fusion member, showing an ability to blend the goals of its base behaviors into coherent actions allow higher order tasks like ramming to occur without disrupting on going attempts to track and shoot and opponent.

### *5.1.5 Discussion*

From the empirical results presented in Table 5.1, the most noticeable grouping is that the fusion based arbitration techniques perform better against the benchmark while the winner-take-all arbiters perform significantly worse against the benchmark.

The reason for the performance gap is rooted in the functional capabilities of the base behaviors. The behaviors made available for this study are atomic in nature and are not intended to be coherent when used alone. Therefore, the arbiters that simulate separate control circuits for each motor command allow continuous tasks like target tracking to continue, while behaviors that affect motion, like ram and wander, compete to provide chassis control. The fusion approach is effective in this experiment, because the efforts of target tracking are usable by the shoot and ram behaviors without having to reproduce the control logic.

The two main limitations plaguing the competitive arbitration techniques are starvation and disruption, which can both be tied back to the functional abilities of base behaviors that they are attempting to support. Starvation occurs when higher priority or behaviors with higher activation levels are continuously active and never yield to the other capabilities present in the structure, thus limiting the global capabilities of the structure. The competitive arbiters uniformly sufferer from disruption because the goals being pursued by individual behaviors stop when control is handed over to higher order tasks that do not continue the lower order tasks that support them. In the case of ram, when the tracking behavior successfully tracks a weaker enemy it overrides the target

tracker intending to charge the opponent, but does not continue the tracking effort and loses the target. The ram behavior then yields control, resulting in an inability to either track or charge the opponent.

The columns in Table 5.1 that classify an arbiter as competitive or cooperative are meant to highlight the split around the need to allow a blending of the behavior recommendations. This is not meant to generalize or suggest that cooperative approaches are always better than competitive ones. In fact, the command fusion arbiter is the only pure cooperative arbiter presented in this study, the others are simply a fine grain implementation of competitive approaches on a per motor command basis. The ranking results only indicate how well each arbiter supports the specific behavior set used in this study and in no way suggest that competitive arbitration techniques are poor in general.

## *5.2 Case Study II: Automatic Discovery of Behavior Structures*

The development of coherent and dynamic behaviors for mobile robots is an exceedingly complex endeavor ruled by task objectives, environmental dynamics and the interactions within the behavior structure. This section discusses the use of the UBF's flexible hierarchical structures using interchangeable behaviors and arbitration techniques [55, 56] to evolve good behavior structures.

Given the number of possible variations provided by the framework, evolutionary programming is used to evolve the behavior design. Competitive evolution of the behavior population is used to incrementally develop feasible solutions for the domain through competitive ranking. By developing and implementing many simple behaviors independently and then evolving a complex behavior structure suited to the domain, this approach allows for the reuse of elemental behaviors and eases the complexity of development for a given domain. Additionally, this approach has the ability to locate a behavior structure which a developer may not have previously considered, and whose ability exceeds expectations. The evolution of the behavior structure is demonstrated using agents in the Robocode environment, with the evolved structures performing up to 122 percent better than one created by an expert.

Mobile robots inherently exist in dynamic environments and are expected to react well in unpredictable situations while performing their task(s). Currently, most robots

employ some form of reactive behavior architecture [44]. To cope with the variety in the environment, agents are implemented with a broad set of skills, or behaviors. The goal of fusing several behaviors into a single complex behavior is to deliver a coherent sequence of actions that are ultimately more effective in a given environment than any single behavior [52]. Such attempts have proven to be a significant endeavor for two reasons. The first is that the code complexity of a behavior grows exponentially as additional traits are added. The second is that development of a behavior that tries to maximize some criteria while minimizing others is the optimization of a multi-objective problem [52].

To ease the complexity of designing and coding a behavior-based system, the ability of the UBF to form arrangements of elemental behaviors into arbitrated hierarchies that are logically, if not semantically, correct. By using this attribute of the UBF and the environment as an evolutionary pressure, an initial population of randomly formed structures is able to organize itself into coherent behaviors that are well suited to combat. Through the repetitive application of ranking each member and then evolving the population by application of a genetic programming algorithm, behavior structures emerge that are effective on an absolute scale [35].

The discussion of this experiment is broken into the following sub-sections: relevant background on evolutionary computation principles, the system's high level design, a detailed explanation of the fitness function and the genetic program, a description of the behavior/arbiter components available for this experiment, a description of the XML behavior representation, the presentation of the results obtained from the evolution of eight independent populations, and concludes with a discussion of the overall experiment.

### *5.2.1 Evolutionary Algorithms Background*

The class of stochastic, global search and optimization algorithms that use the repetitive application of seemingly simple rules to discover emergent behaviors are known as evolutionary algorithms (EA). Such techniques loosely imitate natural evolution and the Darwinian concept of *Survival of the Fittest* [29]. EA techniques are especially effective in large search spaces because, they have a random element that makes them less susceptible to becoming trapped in a local minimum. Since evolutionary

pressures are directing the search, they provide good solutions to a wide range of optimization problems that traditional deterministic search methods find difficult [31].

In nature, the evolutionary process occurs when the following four conditions are satisfied: 1) an entity has the ability to reproduce itself, 2) there is a population of such self-reproducing entities, 3) there is some variety among the self-reproducing entities, and 4) some difference in ability to survive in the environment is associated with the variety [35].

One particular subset of EA algorithms is genetic programming (GP). This subset is defined by its ability to manage the adaptation of complex structures. Typically the structures are hierarchical in nature, stored as trees, rather than sequentially as in genetic algorithms. Since the organization and ordering of a member's structure is important, it must be preserved during crossover (or sexual recombination). A single GP cycle, referred to as an epoch, consists of five major events: 1) a fitness evaluation of each member's ability to cope in the environment, 2) a ranked ordering of the population, 3) a period of recombination where the strongest members have the greatest probability of reproducing, thus propagating successful attributes, 4) an opportunity for mutation, which is optionally used to introduce variety and avoid local minima and 5) a pruning of the population size by removing unfit members. Once one epoch is complete a new epoch begins [19, 36].

Many times an environment is competitive or adversarial in nature, meaning that the members of a population must gain their fitness measure at the expense of another. Such competitive evolutions rank individuals in the population relative to their peers. This approach is beneficial because, despite the members of the initial population being highly unfit, over a period of time, members evolve and rise to higher levels of performance as measured in terms of absolute fitness. What is interesting is that such a process is a self-organizing, mutually bootstrapping process that is driven only by relative fitness (and not by absolute fitness) [35].

### *5.2.2 High-Level Design*

Because the UBF behavior structures are trees, consisting of root nodes with arbiters and leaf behaviors, the mapping to a genetic programming representation is

straightforward. The high-level design of the evolutionary system used to automate the discovery of effective behavior structures is centered on the *fitness function* and the *evolution engine*. An adaptation of the Robocode robot battle simulator, described in section 5.1.1, forms the basis of the fitness function which interacts with the evolution engine via input and output files. Each epoch of the evolutionary process is established by the repetition of four execution stages: Stage I enacts the relative fitness function described in section 5.2.3 to evaluate the relative fitness of individuals in a population. Stage II is the evolutionary engine that advances a population, P(*t*), by one generational time, *t*, i.e. from P(*t*) to P(*t*+1). Stage III enacts the absolute fitness function described in section 5.2.3 to measure a population's current level of fitness, in reference to an unchanging benchmark behavior. Stage IV is a parser that maintains a historical record of each population's evolutionary progress. This four step cycle is shown in Figure 5.5.

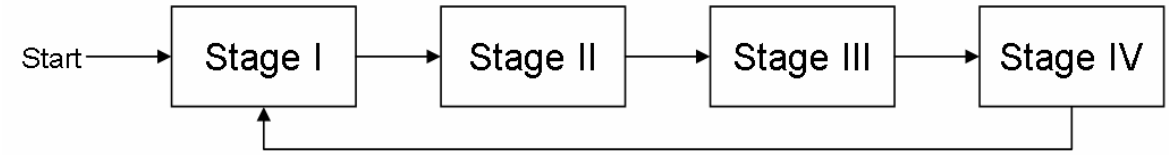



Figure 5.5: Cyclical progression of Stages I through IV.

The details for each of the four execution stages are presented below:

*Stage I*—is the relative fitness evaluation period. The fitness function sets the battlefield conditions using the ten-on-ten melee battle file and configures each combatant with their current behavior structure, which is stored as an XML behavior representation in the *behavior.XML* file. The Robocode simulator plays out a series of engagements and writes a summary to the results file.

*Stage II*—is the evolutionary engine that moves a population from one generational time step to the next. The evolution engine loads the current population, P(*t*), of behavior structures from *behavior.XML* and assess the relative fitness of each member using the results file. The genetic program then generates the subsequent population, P(*t*+1), and concludes by writing the behavior representation of P(*t*+1) to the *behavior.XML* file.

*Stage III*—measures the absolute fitness of the population relative to a fixed benchmark behavior. This evaluation period only provides a reference from which to observe a

population's progress over time and never acts as an evolutionary pressure. The best performing member from the previous generation is measured against the benchmark. The summary of this battle is saved as the results file and is used as input for Stage IV.

*Stage IV*—is a parser that captures the fitness measurement from Stage III and maintains a historical record of a population's progress throughout its evolution.

### *5.2.3 Fitness Function*

The scoring mechanism provided in Robocode provides a quantifiable metric that indicates the relative fitness that two or more behavior structures have in a given environment. In this experiment the fitness function is configurable to operate in either a relative or an absolute fitness evaluation mode. The first is used during Stage I to rank the individuals in a population relative to each other. The second evaluation mode is used in Stage III to capture a population's absolute fitness relative to an unchanging benchmark behavior. This section concludes with a discussion of the noise parameters inherent in using a nondeterministic fitness function and presents the standards for this experiment.

#### *5.2.3.1 Relative Fitness Mode*

The relative fitness mode is the evaluation mode used during Stage I and ranks individuals in the population relative to their peers, regardless of their absolute fitness. The Robocode application is configured using the melee battle file and places ten robots on the battlefield for a twenty-five round, *all-for-one* melee. Because individuals advance their score by exploiting other members, the scores that result from this sequence provide a means of stratifying the members of a population relative to each other. Each member's rating is calculated as the percent difference of a nominal score; values above zero indicate superior combat skills while below zero ratings indicate an inferior level of performance. The probability of selection for an individual is based on their fraction of the total score.

$$(1) \qquad R(k) = \frac{n \cdot score_k}{\sum_{i=1}^{n} score_i} - 1 \qquad\qquad (2) \qquad \Pr(k) = \frac{score_k}{\sum_{i=1}^{n} score_i}$$

An individual's rating and probability of selection are defined by equations (1) and (2) respectively, where $n$ denotes the number of members in a population and $k$ is a specific individual. Equations (1) and (2) are applied to a set of sample data and presented in Table 5.2. Using the sample results in Table 5.2, a nominal score is 3410 and a member with this score earns a rating of 0% and a probability of selection of 0.100. Thus the member Charlie, whose score is 6383, earns a rating of 87% and a probability of selection of 0.187 because its earned score is 87% greater than the nominal score.

Table 5.2: Melee results stratify individuals relative to the other members in a population.

| **Member** | **Score** | ***R(k)*** | **Pr(*k*)** |
|---|---|---|---|
| Charlie | 6383 | 87% | 0.1872 |
| Golf | 4397 | 29% | 0.1289 |
| Delta | 3816 | 12% | 0.1119 |
| Juliet | 3622 | 6% | 0.1062 |
| Bravo | 3214 | –6% | 0.0943 |
| Hotel | 3156 | –7% | 0.0926 |
| Alpha | 2865 | –16% | 0.0840 |
| Echo | 2474 | –27% | 0.0726 |
| Foxtrot | 2114 | –38% | 0.0620 |
| Indigo | 2058 | –40% | 0.0604 |
| **Total** | 34099 | 0% | 1.0 |

*5.2.3.2 Absolute Fitness Mode*

The absolute fitness mode is the evaluation mode used during Stage III to gain insight into how subsequent generations of a population progress over time by ranking against a fixed benchmark behavior. This evaluation is used to observe the fitness of a population on an absolute scale and is never used to drive the direction of the evolution process. In this mode, the Robocode application is configured to set the population's fittest member against the benchmark behavior for a twenty-five round, *one-on-one* battle.

In most cases this approach provides a good estimate of absolute fitness. However, in some cases, a population can discover structures that are particularly good at defeating the benchmark without being a globally optimal solution. For this reason, these values only serve as an indicator of how a population is progressing towards the notion of absolute fitness.

The benchmark behavior, as generated by a user expert, is shown in Figure 5.6 and consists of the behaviors Wander v3, Charge, Dodge and Fire v1 joined by an activation fusion arbiter. The benchmark's observed behavior has three operating modes: one that executes a random S-wander pattern across the battlefield while attempting to track and shoot opponents, another which aggressively charges towards a nearby weaker opponent with guns blazing, and an evasive behavior that emerges above the other two when the benchmark is taking fire from unseen opponents.

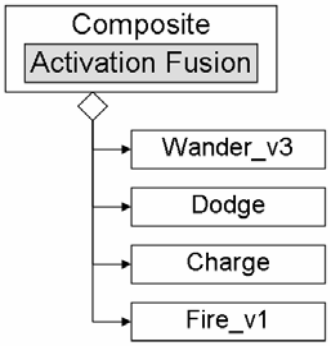



Figure 5.6: The control structure of the benchmark behavior.

*5.2.3.3 Noise Parameters*

The jitter inherent in the absolute fitness function is caused by the stochastic variance in the simulator's ability to accurately stratify members relative to each other. The nondeterministic progression of battles in Robocode is caused by random starting postures and the dynamic interaction of opposing behavior algorithms. The results of any one battle have some level of uncertainty, with the more rounds per battle, the smaller the uncertainty. To demonstrate this, a sequence of battles is created with the benchmark facing itself in combat. On average, when identical behavior structures are set against each other, neither one should score better than the other. When battles consist of five rounds each, the average relative fitness measured is 0.6% with a standard deviation of 40.2%. When battles consist of twenty-five rounds each, the average relative fitness is 0.5% and the standard deviation drops to 17.6%. These results are shown in Figure 5.7 as (A) and (B) respectively.

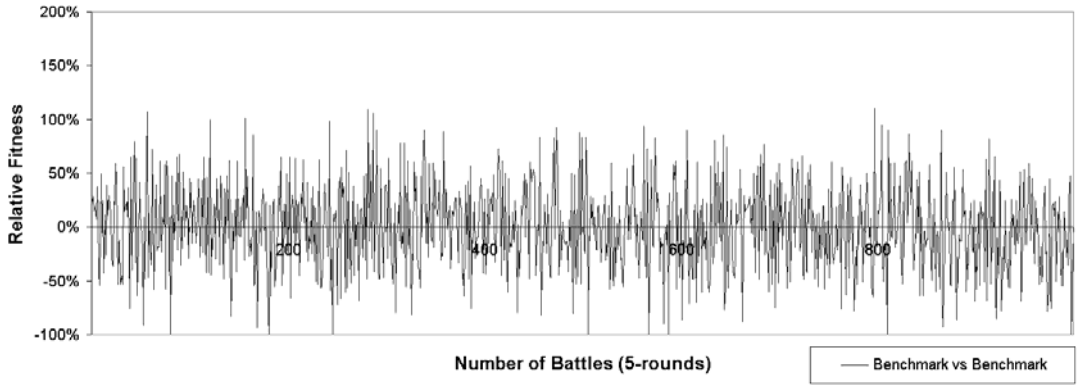



Figure 5.7a: Noise for Benchmark vs. Benchmark (5-rounds).

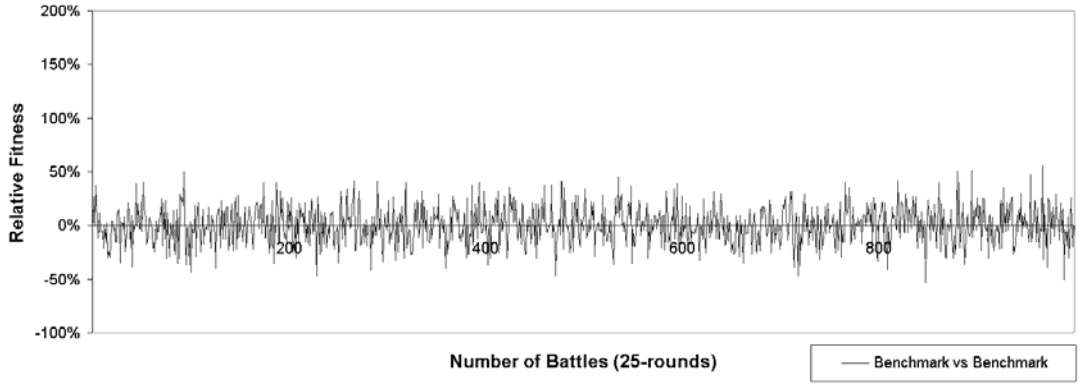



Figure 5.7b: Noise for Benchmark vs. Benchmark (25-rounds).

Although increasing the number of rounds per battle reduces jitter and more accurately stratifies an individual's relative fitness, this approach is prohibitive due to time requirements. To keep the speed of the evolutionary cycles manageable, twenty-five round battles are established as the standard for this experiment, setting the fitness function's noise parameter at plus or minus 17.6% per battle. To provide a cleaner representation of how sequences of battles are progressing, a ten-tap moving average is applied to smooth the results and establish a noise floor. Applying this filter to the data in Figure 5.7b establishes a noise floor expectation with a near zero average and a jitter of 5.45%. The effect of using this approach is illustrated in Figure 5.7c.

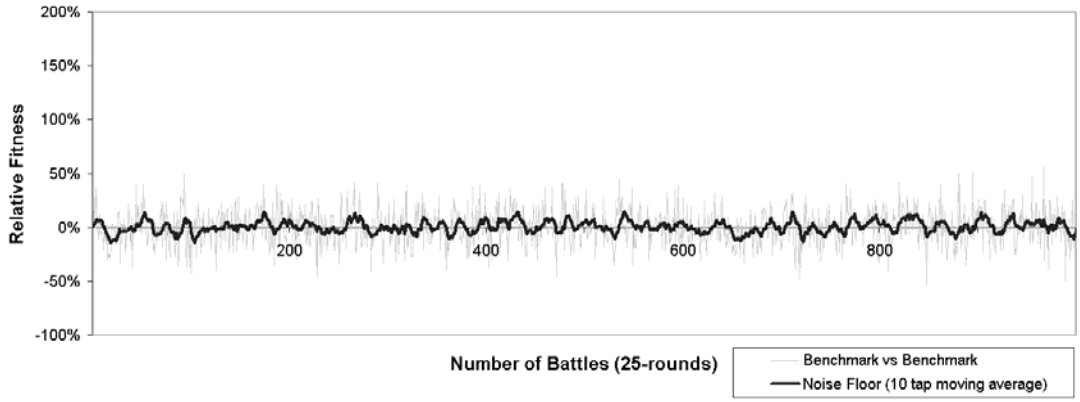



Figure 5.7c: Applying a 10-tap moving average dampens variance, shows trends over time and establishes the experiment's noise floor.

### *5.2.4 Genetic Program*

The hierarchical nature of behavior structures under the UBF allows a genetic program (GP) to perform a stochastic search of the solution space. The GP in this experiment maintains a fixed population of ten members and uses *elitism*, *mutation* and *generational recombination* to guide the search from an initial random population towards a set of behavior structures that are coherent for the domain. The GP's parameter settings are specified in Table 5.3.

Table 5.3: Parameter Settings for the genetic program.

| Parameter | Symbol | Setting |
|---|---|---|
| Population Size | $n$ | 10 |
| Elitism Rate (%) | $E$ | 10% |
| Mutation Rate (%) | $M$ | 10% |
| Generation Rate (%) | $G$ | 80% |
| Contributing Set Size | $r$ | $G \cdot n$ |
| Variance (%) | $v$ | ± 10% |
| Max Branching | $b$ | 4 |
| Max Depth | $d$ | 7 |
| Number of Generations | $X$ | 1000 |

The Elitism rate *(E)* provides the GP a means of propagating successful structures as they are discovered. By advancing a fraction of the population with highest fitness directly from population P($t$) into P($t$+1), the GP partially becomes hill climbing.

The Generation rate *(G)* specifies the rate of generational recombination. This fraction of the population P($t$+1) are new behavioral structures formed by the crossover

of members in the contributing set. Recombination is a two step process consisting of a *selection* step and a *crossover* step:

The selection process uses stochastic universal selection (SUS) [5] to choose the contributing members from the population P(*t*). SUS uses *r* equally spaced markers across the population's score distribution. The selection markers shift within the selection space based on the initial value (or seed). The seed is a randomly selected value between zero and 1/*r*. Using the sample results in Table 5.2 and a seed of 0.0825, a selection process is shown graphically in Figure 5.8 where the application of SUS chooses the individuals Alpha, Charlie, Charlie, Delta, Echo, Golf, Hotel and Indigo to form the set of contributing members. SUS is used over fitness proportional selection in an attempt to avoid premature convergence of the population by allowing successful individuals a good opportunity at selection while still giving less successful members an opportunity to contribute their genetic material to the next generation.

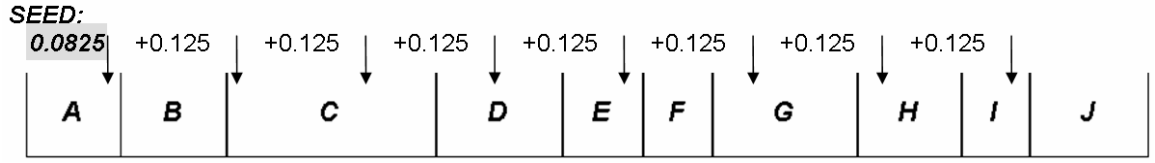



Figure 5.8: Eight members are selected from the current population using SUS across the score distribution and a random seed of 0.0823 to form the contributing set.

During crossover, pairs of individuals are randomly selected from the contributing set of members and through the process of genetic recombination, each pair forms two new individuals that are ultimately introduced into the population P(*t*+1). The crossover process is illustrated in Figure 5.9. During a crossover event, a randomly selected branch is removed from each contributing member and given to the other. The portion received is placed at the crossover site. By swapping behavioral substructures, the two offspring are unique structures, but are a derivative of their parent's attributes.

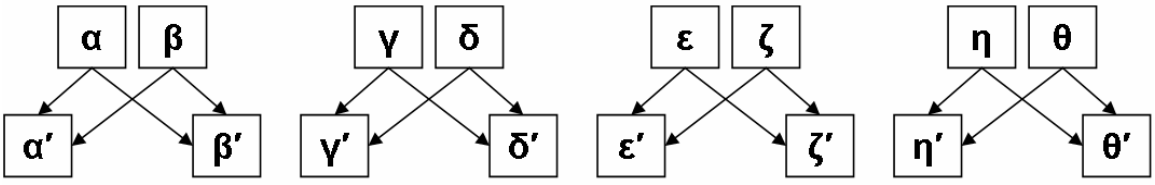



Figure 5.9: The contributing individuals are randomly paired to form four crossover events. The eight subsequent individuals become members of the population P(*t*+1).

Before being added to the population P($t$+1), the resulting offspring are pruned at the maximum depth *(d)* to limit their complexity and are given additional variation *(v)* through fluctuations in the behavior weightings held by each arbiter. The new generation of members is then introduced into the population P($t$+1).

The Mutation rate *(M)* specifies the fraction of the population P($t$+1) that are formed as randomly generated behavior structures. The addition of the random members to the population maintains the genetic diversity of the population and promotes exploration throughout the course of the search.

### *5.2.5 Description of Elemental Components*

Using the UBF interface, thirteen elemental behaviors and seven arbiters are developed and tested as independent components. The functionality of each component is briefly described below and then used as the pool of genetic material from which members of the population are formed. The behaviors are:

*Charge*—when another robot (with a lower energy level) is detected, this behavior causes our robot to turn towards the other and charge towards it, attempting to cause damage by hitting it.

*Dodge*—when hit by a bullet or by another robot, this behavior causes our robot to respond with an evasive maneuver based on the type of attack and afflicted quadrant.

*Fire v1*—has three operating modes. When no target is detected, the default mode turns the turret in a clockwise direction. When a target is detected, the target tracking algorithm causes the gun turret rotation to slow or reverse its direction in an attempt to continue tracking the target. In addition to target tracking, when the target is less than three degrees off boar site our robot will fire on another, the power committed to the bullet is reduced as a function of the target off boar site angle.

*Fire v2*—is exactly like Fire v1 with the exception that the maximum power is always committed to the bullet.

*Return Fire*—holds a grudge against another that has previously attacked our robot. When no specific target is set, the default mode behaves exactly like Fire v2

until our robot is shot or hit by another. When an aggressive opponent is specified, only that target is engaged. The aggressor remains the target until it is killed.

*Scan Left*—turns the gun turret and the radar counterclockwise.

*Scan Right*—turns the gun turret and the radar clockwise.

*Short Range Fire*—is based on Fire v1, but only fires at targets that are at close range and are less than fifteen degrees off boar site. Maximum power is always given to the bullet.

*Sitting Duck*—will always recommend that our robot stop all motion, including the motion of the gun and the radar.

*Sniper Fire*—is adapted from Fire v1 and is specialized to attack slow moving targets at long ranges. When a target is found to be stopped or moving slowly it recommends that our robot stop its movement and track the target until it is less than one half of a degree off boar site. Maximum power is always given to the bullet.

*Wander v1*—circumnavigates the perimeter of the board. Our robot's current velocity is maintained unless it is less than the minimum.

*Wander v2*—simulates Brownian motion by randomly executing a series of fifty degree arcs. When a wall is detected, the current velocity is flipped to reverse our direction.

*Wander v3*—performs a series of "S" turns. Random selection is used to set the length of the arc to be between thirty and one hundred twenty degrees before changing the turn direction. When a wall is found, the current velocity is reversed to change our direction.

The available arbitration techniques are:

*Activation Fusion*—is a semi-cooperative arbiter that uses a highest activation selection approach on a per motor command basis. Unlike highest activation, activation fusion builds a new action set, allowing the motor commands left unspecified by the behavior with highest level of activation to be set using the recommendations of behaviors with lower activation levels. When used with

market based systems, this technique is easily referred to as utility fusion, but risks confusion with Rosenblatt's utility fusion [44] behavior architecture.

*Command Fusion*—is derivation of the motor schema architecture [2], a cooperative arbitration approach that uses summation and normalization of proposed motor commands to derive the resultant set of motor commands. The input of all contributing behaviors are used on a per motor command basis to form the resultant command vector.

*Highest Activation*—is a winner-take-all arbiter that returns the action set with the highest vote value. Inspired by the action-selection architecture [37], this approach provides a dynamic mechanism for competitive selection by allowing behaviors to indicate their urgency for activation. Associated behavior weights are used to internally tune global performance by scaling the votes of behaviors that either over or under vote. The concept of activation levels is synonymous with the concept of utility in market based systems.

*Highest Priority*—is a winner-take-all arbiter that returns the action set of the highest priority behavior indicating a desire to act, regardless of vote value. Like Subsumption [15, 17], the recommendations of lower priority behaviors only execute if higher priority behaviors abstain.

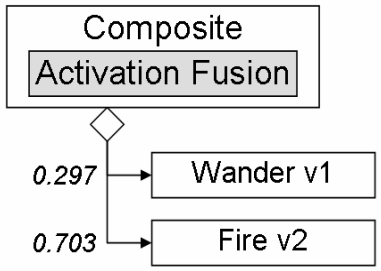
*Monte Carlo*—is a stochastic arbitration technique that uses fitness proportional random selection to activate one sub-behavior for a period of time. At the end of the period another random selection occurs, activating the chosen sub-behavior for the current period.

*Null Arbiter*—always passes an empty action back, regardless of the action set passed in. Using this arbiter deactivates the branch of control where it is applied.

*Priority Fusion*—is a semi-cooperative arbiter that uses priority based arbitration on a per motor command basis. Unlike the highest priority arbiter above, priority fusion builds a new action set that allows the unspecified action fields of higher priority behaviors to be filled by lower priority action requests.

### *5.2.6 XML Behavior Representation*

The tree structure of behaviors under the UBF allows them to be represented directly using extensible markup language (XML). In general, XML is a text based file that uses a structured language format to communicate information and is typically used to support interoperability between independent systems. In this experiment, an XML representation of the current behavior population is stored in the *behavior.XML* file. An example of a behavior structure encoding is given in Figure 5.10.




```
<member name="Alpha">
   <node composite="Y">
     <arbiter w1="0.297" w2="0.703">
              ActivationFusion</arbiter>
     <node>Wander_v1</node>
     <node>Fire_v2</node>
   </node>
</member>
```

Figure 5.10: Example of a behavior structure and the corresponding XML encoding.

This experiment uses the XML behavior representations to allow changes made by the evolution engine to configure the fitness function. In Stage I, the fitness function configures the battle by placing the robots Alpha through Juliet on the battlefield. Each robot then request their behavior from the behavior factory [25]. Within the factory are mechanisms for parsing the *behavior.XML* file and reconstructing a behavior structure from its XML representation. Additionally, the use of XML allows a representation of the current population to be continuously available in a persistent state, allowing an evolutionary process to be started, stopped and re-started at will while limiting the risk of loosing computational progress to a single epoch.

### *5.2.7 Results*

In this experiment, eight behavior populations are independently evolved over the course of 1,000 generations. While the initial populations are collections of randomly generated behavior structures and are generally unfit on an absolute scale, they introduce variety into the population. Through the repetitive ranking, selection and recombination of the members within a population, initially random structures organize themselves into populations of structures that are measurably effective on an absolute scale [35].

In this experiment each of the eight initial populations converges on relatively simple solutions that exploit the homeostatic aspects of the Robocode domain [3]. This section discusses how the populations' absolute fitness progresses over time, then discusses the critical aspect of the Robocode domain that acts as the evolutionary pressure shaping the solutions, and finally concludes with a comparison of how the individual solution structures rate relative to each other.

The absolute fitness of each population is a measurement of the population's performance against the fixed behavior structure, which allows the progress of independent evolutions to be compared directly. The fitness rating is calculated as the percent difference of a nominal score; values above zero indicate superior combat skills while below zero ratings indicate an inferior level of performance. The trend graph presented in Figure 5.11a shows the fitness of eight populations as they evolve over time.

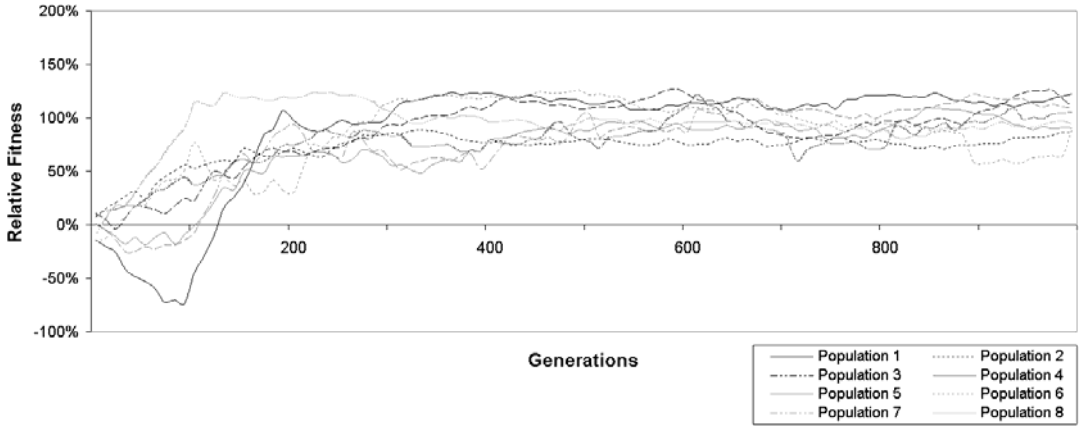



Figure 5.11a: Progression of eight populations as measured relative to the benchmark.

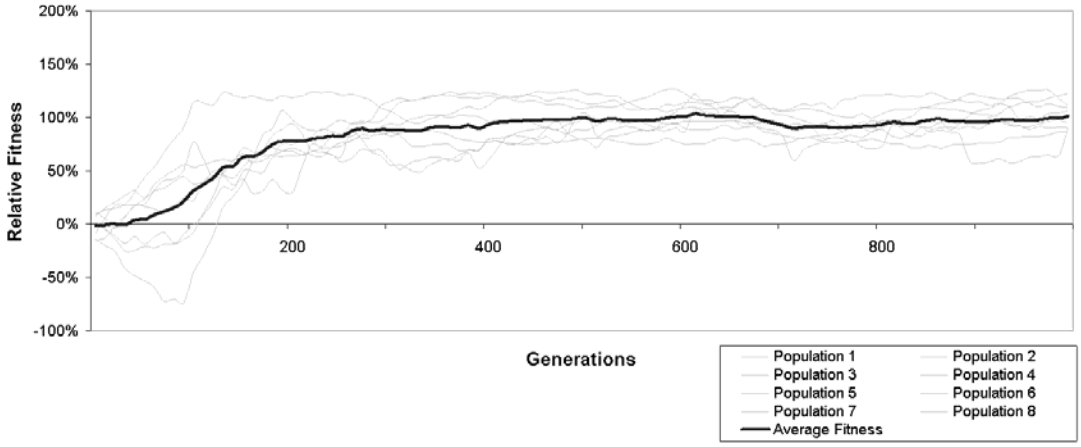



Figure 5.11b: Progression of average fitness as measured relative to the benchmark.

The use of a fixed benchmark behavior to evaluate absolute fitness is somewhat misleading because it allows configurations that are exceedingly effective against the benchmark to achieve high fitness ratings without being an effective solution in general. This anomaly presents itself during run eight which initially favors a configuration that displays a high level of fitness against the benchmark (see generations 100 through 300 in Figure 5.11a), but later abandons that family of configurations in favor of structures that are more successful in general. To reduce the affects of such anomalies and achieve a better indication of how the populations are progressing towards absolute fitness, the average progress of the eight populations is used. Figure 5.11b presents the average progress of the eight populations as measured against the benchmark.

Looking at the progression of average fitness during the course of one-thousand generations, a notable period of improvement occurs during the initial two-hundred generations where fitness improves from a nominal rating to a rating of 78%. The remainder of the evolution is relatively stable, maintaining an average rating of 94% against the benchmark and ends with a rating of 101%. A progression of the absolute fitness using discrete time steps is presented in Table 5.4.

Table 5.4: Progression of absolute fitness during discrete intervals of 100 generations.

| Generations | Minimum Rating | Average Rating | Maximum Rating |
|---|---|---|---|
| 1 – 100 | -1% | 6% ±8% | 21% |
| 101 – 200 | 31% | 57% ±16% | 78% |
| 201 – 300 | 77% | 83% ±5% | 89% |
| 301 – 400 | 87% | 89% ±18% | 92% |
| 401 – 500 | 93% | 97% ±2% | 99% |
| 501 – 600 | 97% | 98% ±14% | 101% |
| 601 – 700 | 95% | 100% ±3% | 104% |
| 701 – 800 | 90% | 91% ±1% | 93% |
| 801 – 900 | 93% | 96% ±2% | 98% |
| 901 – 1000 | 95% | 98% ±2% | 101% |

While the evolution of eight independent populations converges on a variety of solutions, each structure captures a similar aspect of the Robocode domain. The populations naturally move towards somewhat passive solutions that are capable of attacking a target when conditions are favorable. This approach is effective because a robot must commit a fraction of its energy when shooting at an opponent. Like gambling,

it benefits a robot to shoot when there is a reasonable expectation of hitting a target. If the shot misses, the committed energy is lost. If the shot hits a target, the target's energy is reduced by that amount and the shooter claims twice the energy committed. Observations made during the fitness evaluations in Stage III show that the aggressive nature of the benchmark behavior is self-defeating because it often fires from long distances where there is little expectation of scoring a hit. The more conservative behavior allows members to achieve high relative fitness ratings by simply evading the benchmark until it cripples itself by draining its own energy reserves.

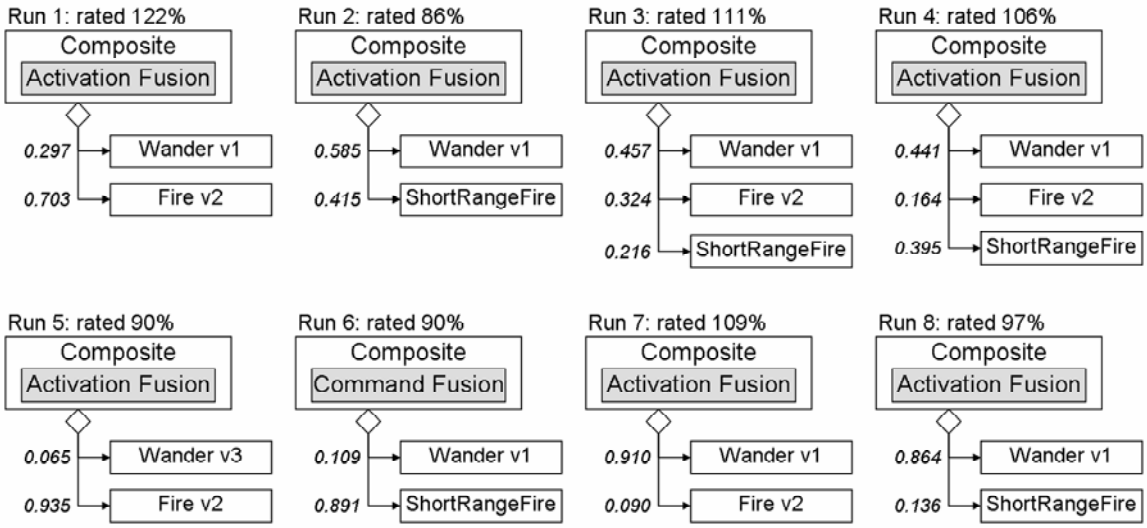



Figure 5.12: Behavior structures discovered from the evolution of eight randomly generated behavior populations.

The solution structures discovered by each of the eight populations are shown in Figure 5.12. At first glance, the common thread between the solutions is that they each employ a motion behavior and a tracking/shooting behavior joined by a fusion based arbiter. The use of a fusion based arbiter allows the robot to pursue multiple objectives simultaneously.

Conspicuously missing from the solutions above are the shooting behaviors: *Return Fire*, *Fire v1* and *Sniper Fire*. Having identified the importance of using a more conservative shooting approach, *Fire v1* and *Return Fire* are undesirable because they impose no range restriction and take unlikely shots at distant targets. The *Sniper Fire* behavior, a highly specialized behavior for shooting unmoving targets at long range, is

likely to become obsolete because a population adopts continuous motion as a minimal requirement for survival.

Of the motion based behaviors, *Wander v2*, *Charge* and *Dodge* each fail to make an appearance in the solution set. *Wander v2*, which simulates Brownian motion, was intended to produce erratic movements that can not be effectively tracked by an opponent. In reality, it produces erratic motion in a localized area, making shots in the general direction more likely to score a hit. As noted above, somewhat passive behaviors are able to conserve their energy and achieve higher mortality rates, thus a behavior, like *Charge*, that moves our robot into an opponent's effective radius is also unfavorable. The absence of the *Dodge* behavior suggests that an ability to sustain continuous motion can act as a passive means of evading incoming attacks and indicates that such defensive measures are "good enough."

Observations of the solution structures in Figure 5.12 during battle shows that each is coherent, meaning that the behavior has the ability to perform basic elements of combat like tracking and shooting targets while moving within the battlefield without impeding its own progress towards the immediate goal and is able to consistently demonstrate a level of fitness that is superior to the benchmark. The real question is, "How good are these solutions on an absolute scale?"

To better understand how the eight solutions rank on an absolute scale, the eight solutions are compared in an eight-on-eight battle to discover the fitness of each solution structure relative to the others. This approach uses a series of 250 battles to create an inter-population fitness evaluation and the results are shown in Figure 5.13. Rather than separating into bands, where some solutions consistently achieve higher performance ratings than others, they are (with the exception of population 5) tightly interwoven, indicating that the solutions presented by the individual evolutions are equally matched. With a performance variance equal to the noise floor, seven of the resulting behavior structures are considered to be equivalent solutions.

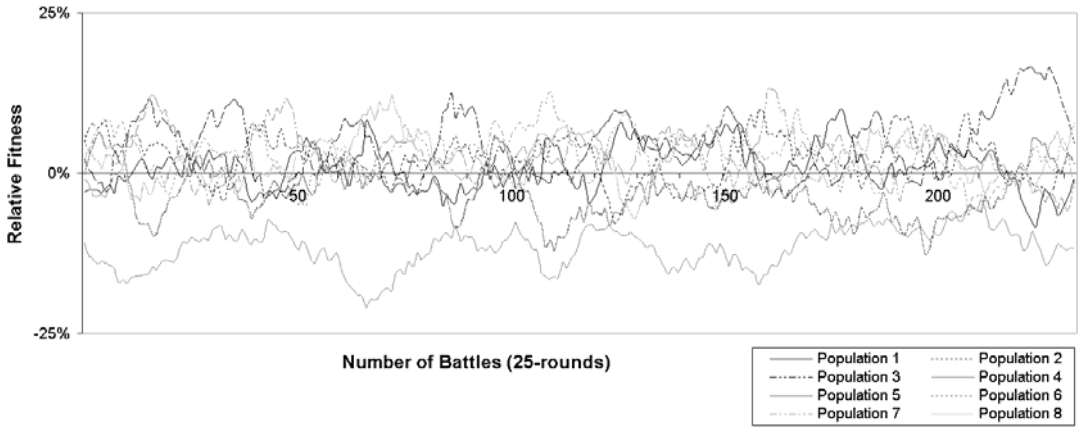



Figure 5.13: Relative fitness of the eight population runs, where seven of the solutions are considered equivalent. The outlier is run 5.

The solutions presented by each run are relatively simple structures, lacking the depth and complexity typically associated with genetic programming solutions. Each solution structure presents a clear pairing of one motion behavior with one or two shooting behaviors. The lack of multiple skills within successful structures indicates that the scope of the elemental behaviors is too large. The behaviors provided, while incomplete for the domain, prefer to act alone and do not act as generic operators that can be composed by an EA to form deeper and more intricate solution structures that have coherent outward operations.

### *5.2.8 Discussion*

The ability of the unified behavior framework to simplify the development and testing of behaviors for a given domain is demonstrated through the use of a genetic program to automate the discovery of effective behavior structures from a pool of simple behavior and arbitration elements. In this experiment, a genetic program is used to discover combinations of elemental components that contribute to the robots motion and its ability to track and shoot targets. The ability of the UBF to support the composition and recombination of behavior structures by the genetic program validates its ability to form structures that are logically correct, if not semantically coherent for a given domain.

In robotic behavior-based system development, the optimal solution is generally unknown and potentially changes with the introduction of new components. Along with the broad capabilities of the UBF, the use of a stochastic search discovers good solutions

and is recommended as a useful tool for developing behavior-based systems. The results show that this method is more effective than relying on raw human cleverness to achieve an optimal configuration directly. Additionally, the close relative fitness of the solution structures indicates that many equivalently good solutions exist within a domain, and that the approach is feasible for other robotic domains.

## *5.3 Case Study III: Real-Time Behavior-Based Controller*

Autonomous systems that operate in the real-world have an inherent requirement to be both robust and responsive to sudden and unpredictable changes in the environment. Typically, reactive behavior routines are tasked with maintaining the safe operation of the system and, as demonstrated by YARA [18], the ability of these low-level routines to run at periodic intervals is crucial to the safety and reliability of the robot's operation. The need to make some processes "more important" than others is becoming common in applications where responsiveness is measured in milliseconds of delay. This section presents the UBF in a goal directed configuration performed using a Pioneer P2-AT8 robot running RTAI [38] beneath a standard Linux [49] installation and an adaptation of the Player control suite [27]. This implementation demonstrates that the system is able to maintain a stable basis of reactive-control with time-critical tasks responding with less than 100 μs of delay, regardless of the system's computational load. For this study, the computational burden normally imposed by predictive and deliberative elements are simulated using ten continuous ping floods to the local host address. By establishing the robot's reactive controller as a set of real-time processes, the routines that update the State and form the goal directed behavior execute at established intervals that are unaffected by the adverse computational loads that are disruptive to other concurrent processes competing for processing time within the Linux user-space.

The discussion of this experiment is broken into the following sections: the system's high-level design, an explanation of the UBF integration with the Player control suite to form a responsive behavior-based controller, a description of the goal directed behavior structure, and concludes with the experiment results and a discussion of the experiment overall.

### *5.3.1 High-Level Design*

The Pioneer P2-AT8 is a four-wheeled robotic platform equipped with 16 sonar sensors to sense obstacles and a dead reckoning navigation capability. An ability to schedule the robot's low-level control routines as periodic real-time tasks is provided by hooks into the RTAI microkernel while all other processes execute in the user-space of a typical Linux environment. In this experiment, elements of the Player control suite are used to establish a behavior-based controller based on the UBF design presented in Chapter IV. To give this behavior-based controller the ability to provide a responsive basis of reactive control, independent of fluctuation in the system's computational load, the controller's subcomponents are established as hard real-time tasks that bypass the Linux scheduler and run in the context of the RTAI scheduler. A block diagram of the high-level design is presented in Figure 5.14 and shows how RTAI resides directly above the Pioneer hardware and that the behavior-based controller processes are able to bypass Linux and be treated as real-time processes by the RTAI scheduler.

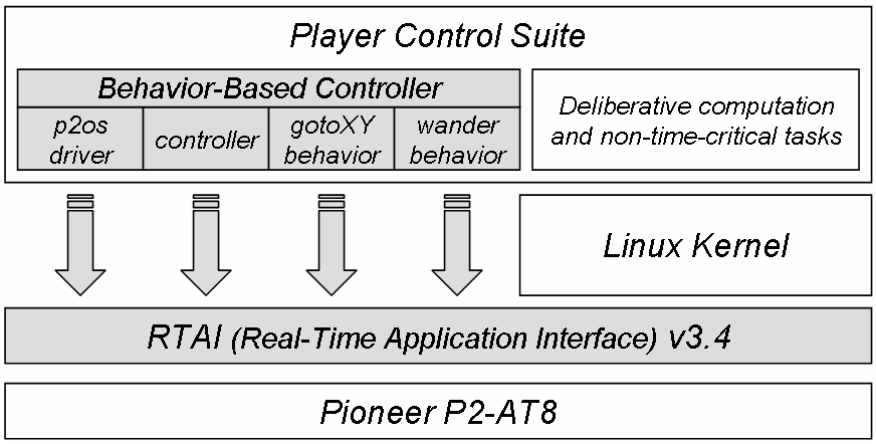



Figure 5.14: Real-time Player tasks bypass Linux and run on the RTAI scheduler.

The behavior-based controller is made responsive by allowing its subcomponents to preempt the Linux environment and execute at assigned intervals. The *p2os_driver* and *controller* components are modifications of the Player control suite while the *gotoXY* and *wander* modules are implemented using the threaded behavior design presented in section 4.7. These two behaviors are joined by a *higest_activation* arbiter to form the goal directed behavior used by the controller. By implementing these four execution threads as hard real-time tasks, a basis of control is established that guarantees less than 100 µs of

delay, regardless of the computational load that exists in the Linux user-space. Table 5.5 presents the scheduling specifications for this design along with the priorities and periodic execution rates for each task. The *p2os_driver* provides a central service by ensuring that the *State* correctly represents the current environment. If the *State* falls out of sync with the real-world, the remaining controller components become ineffective, consequently the *p2os_driver* task holds the highest priority and executes at 10 ms intervals. The elemental behaviors are given the next highest priority because their evaluations of the *State* form the basis of what actions the *controller's* active behavior will recommend at any given time. The final consideration is that the execution rate of the *controller* is set to request an action recommendation at twice the rate of the fastest elemental behavior, an approach based on the Nyquist sampling rate [42]. Because simple periods are used, harmonics exist that require some processes to run at exactly the same time. To reduce unnecessary latency due to scheduling collisions, the absolute execution time of each task is staggered using offset values that cause the tasks to interleave their execution times.

Table 5.5: Scheduling configuration for real-time tasks.

| Task | Description | Offset | Period | Priority |
|---|---|---|---|---|
| *p2os_driver* | Pioneer HW Interface | 10 ms | 10 ms | 1 |
| *Controller* | Behavior-Based Controller | 0 ms | 100 ms | 3 |
| *gotoXY* | Elemental Behavior | 20 ms | 25 ms | 2 |
| *Wander* | Elemental Behavior | 30 ms | 25 ms | 2 |
| Linux | The Linux Environment | idle | | 9999 |

### *5.3.2 Player Adaptation*

The modification to the Player control suite [27] that forms a responsive behavior-based controller is two fold: The first modification is that a behavior-based controller is established by replacing the *clientproxy* concept with thread-safe versions of the standard UBF *State* and *Action* interfaces. The second modification allows the Pioneer drivers to switch into a hard real-time mode, making them schedulable as priority tasks that preempt the Linux kernel when they enter a ready to run state and register with the RTAI hardware abstraction layer to interface with the hardware components in real-time.

The low-level control loop of the behavior-based controller consists of the continuous execution of the three-step process presented in Figure 5.15. First, the *State* is

updated by the *p2os_driver* to represent the current conditions of the environment. Second, the active behavior is asked to generate a recommended action by invoking the *genAction* method. Finally, the proposed action is given the authority to issue motor commands to the driver via the *execute* method, closing the sensing/action control loop of the low-level controller.

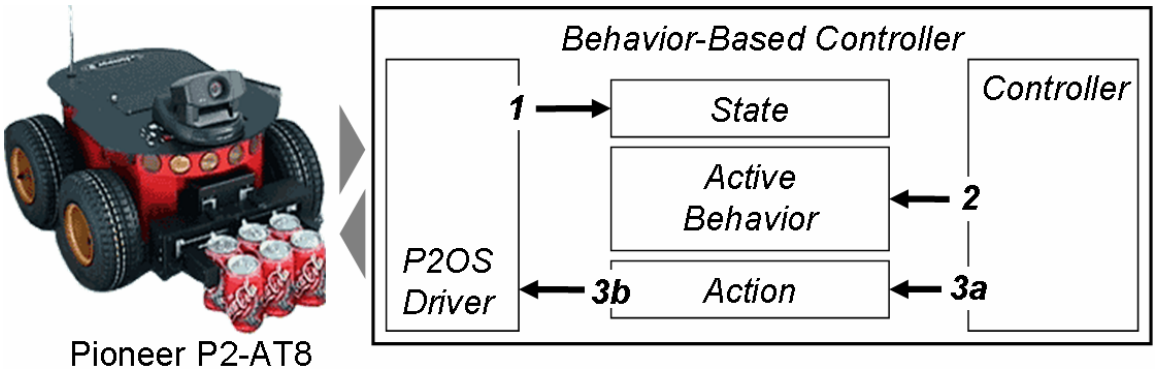



Figure 5.15: A behavior-based controller modeled after on the UBF. A low-level control loop is established a three step process (1) update the state; (2) generate an action recommendation; (3) authorize the action to enact motor commands on the robot.

Under this behavior-based controller design, the set of behaviors assume that the *State* is representative of the current environment. This assumption places the responsibility of keeping the system in sync with the real-world onto the set of real-time drivers, because they are the routines that update the *State* with current sensor data. The ability to establish a driver as a real-time task allows its routines to execute at predictable intervals driven by the sensors update rate, which in turn ensures that the central *State* is updated at regular intervals.

The next responsibility of the behavior-based controller is to generate an action recommendation based on the current conditions of the environment. By following the model of the UBF presented in Chapter IV, the controller keeps an active behavior without knowing about its implementation. In this design, the *Server* class is taken from the Player control suite and modified to form the real-time controller. Unlike the three-step process presented in Figure 4.4, the *State* is updated asynchronously, and the controller assumes that the *State* is an accurate representation of the current environment. Thus, the controller enacts a two-step periodic process that first requests an action recommendation from its active behavior and then authorizes the action to enact the

recommended motor commands on the robot. This process is shown by the following pseudocode:

```
while(running) {
  action = activeBehavior.genAction(State);
  action.execute(Robot);
  rt_wait_period();
}
```

The *controller*, like the *p2os_driver*, is established as a real-time task, allowing the active behavior to evaluate the environment at predictable intervals. An ability to regularly update and evaluate the environment allows the robot to operate in a safe and dependable manner by remaining responsive to changes in the environment.

Following the UBF model, the *State* and *Action* classes are introduced as generic interfaces to the *p2os_driver*. The central *State* object is a representation of the current environment and includes decoupled sensor data, positional information, goals, and current operational parameters. Explicitly missing from the *State* are methods that access the *p2os_driver's* motor command interface. This capability is embedded in the *execute* method of the *Action* class, and requires a reference to the robot's *p2os_driver*. This requirement ensures the coordinated operation of the robot by allowing the *controller* to enact the action recommendation returned by the active behavior on the robot. The bifurcation of the *p2os_driver* into two interfaces allows the UBF to make information about the robot's current state widely available while protecting against behaviors that may act unilaterally on the robot.

The real-time processes *gotoXY* and *wander* are used to form the behavior-based controller's goal directed behavior, which is presented in detail in the next section.

### *5.3.3 Goal Directed Behavior*

A goal directed behavior is shown in Figure 5.16 and is established using a control structure that includes a goal-seeking behavior and a random-wander behavior joined by a highest activation arbiter. The goal-seeking element directs the robot along a direct route to a goal location specified in the shared *State*. The random-wander behavior provides a means of obstacle avoidance. The use of a highest activation arbiter allows the goal seeking component to yield to the random-wander behavior for a period of time in

an attempt to bypass an obstacle in the path toward the goal and then out vote it to make another attempt at moving towards the goal. Figure 5.17 shows the robot's observed path.

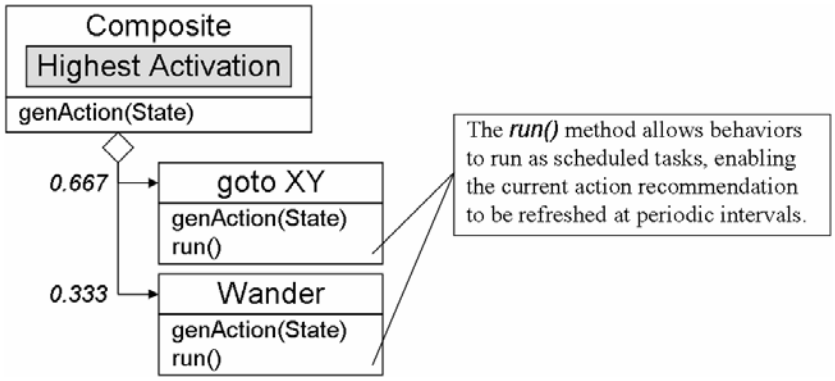



Figure 5.16: Control structure of a goal directed behavior formed from a goal-seeking element and a random-wander element joined by a highest activation arbiter.

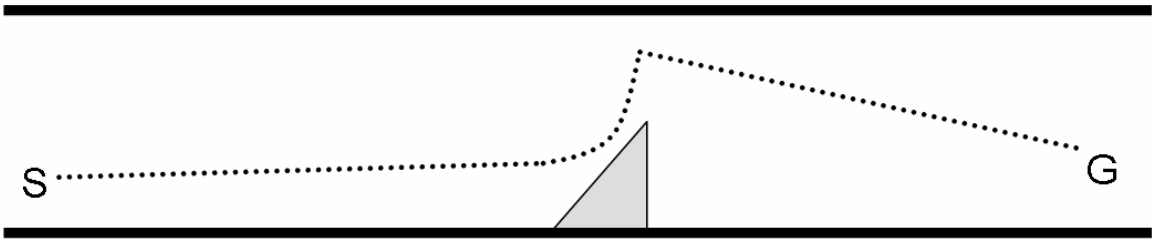



Figure 5.17: The observed path of the Pioneer P2-AT8 robot as it navigates a horizontal hallway from its starting location (S) to the target goal location (G). The shaded triangle represents the mid-course obstacle obscuring the robots path to the goal.

The implementations of the elemental behaviors used in this experiment are based on the concurrent and real-time design presented in section 4.7. Each behavior maintains an action recommendation that is accessible by calling the *genAction* method. The evaluation logic that builds the action is moved into the *run* method and executes as a periodic real-time task to keep the behavior's current action recommendation relevant to the current environment.

At its highest level, a goal directed action recommendation is available via the *genAction* method. When an action recommendation is requested, the composite node builds a set of action recommendations by calling *genAction* on each its sub-behaviors. The set of actions is then evaluated by the arbiter to form a single action that is subsequently returned as the goal directed behavior's current recommendation.

### *5.3.4 Results*

The results of this experiment demonstrate that a responsive basis of control is attained by implementing time-critical routines as real-time tasks. This experiment establishes a responsive basis of control using four periodic processes that are established as real-time tasks (i.e. *p2os_driver*, *controller*, *gotoXY*, and *wander*) and demonstrates the ability of these routines to execute at predictable intervals regardless of the computational load within the Linux user-space. The four routines that form, the behavior-based controller are instrumented to capture the current time and calculate the latency experienced per execution period. Latency is measured as the time between when a periodic task is scheduled to execute and when it actually beings executing. For example, if a task is scheduled to execute every 20 ms and the difference between the previous time hack and the current time hack is 22 ms, the reported latency is 2 ms. A process's jitter is evaluated by making a series of latency measurements over time.

The latency measurement achieved by this experiment far exceeds the 100 µs hard real-time guarantee provided by the RTAI documentation. The empirical results of this experiment indicate that the periodic scheduler executes tasks early, as indicated by the negative latency values in Table 5.6, and is predictably consistent on the order of ±1 ns. The jitter observed jitter for each real-time task is shown by the graphs in Figure 5.18.

Table 5.6: Latency statistics for real-time tasks.

| Task | Latency | | Maximum |
|---|---|---|---|
| *p2os_driver* | –1.5 µs | ±1.0 ns | 998 µs |
| *controller* | –15.1 µs | ±1.0 ns | 984 µs |
| *gotoXY* | –3.0 µs | ±1.0 ns | 996 µs |
| *wander* | –3.0 µs | ±1.0 ns | 996 µs |

The latency measurements taken and the observed jitter for each task indicate that critical routines can be scheduled to execute at predictable intervals by removing them from the context of the Linux environment and running them as real-time processes. The periodic 1000 µs latency spikes have been linked to the RTAI periodic task scheduler and a bug report has been submitted to the RTAI project development team.

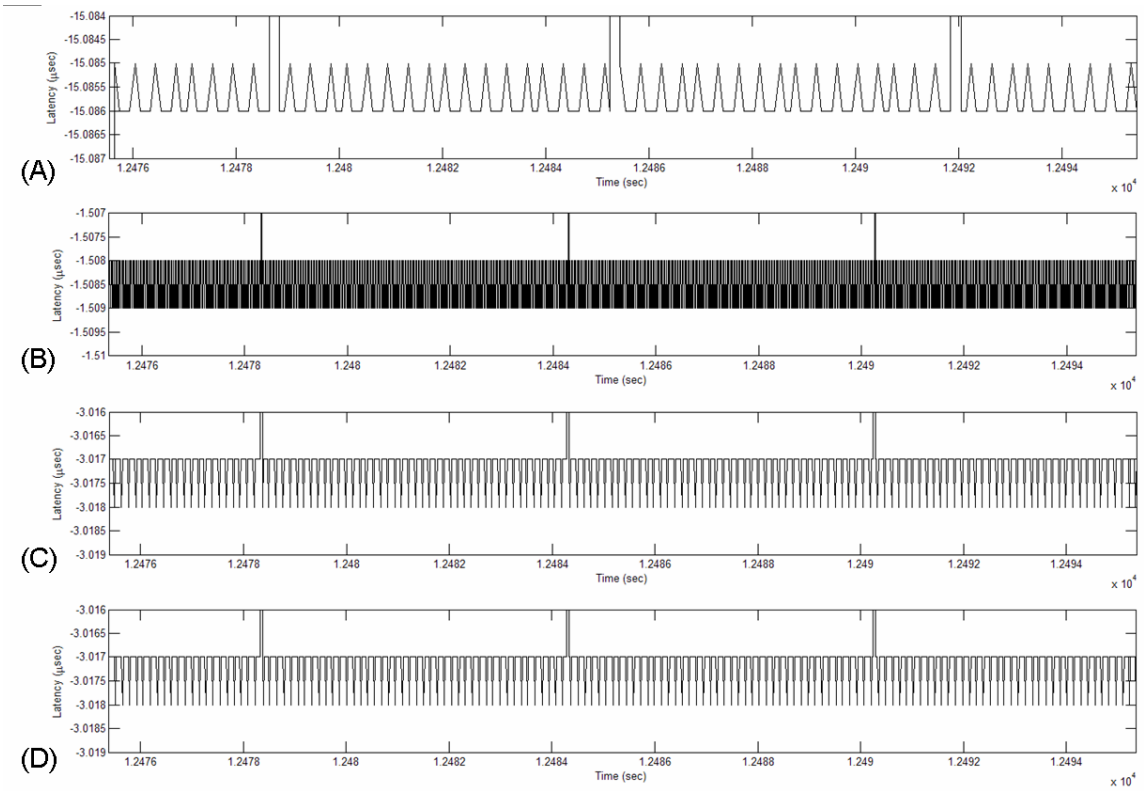



Figure 5.18: Observed latency jitter for (A) the *controller* task; (B) the *p2os_driver* task; (C) the *gotoXY* task; and (D) the *wander* task.

### *5.3.5 Discussion*

Mobile robot architectures are a mixture of interconnected processes working to achieve specific results. With the speed of modern processors and the ability of operating systems to manage multiple threads of execution, many robot architectures are implemented using single processor systems. Research in mobile robotics and autonomous systems are also finding an increased need for process scheduling that is predictable and accurate in relation to the real-world. Mapping and navigation are examples of routines that are sensitive to unexpected latencies of more than one or two milliseconds.

This experiment demonstrates that the ability to establish low-level control routines as real-time tasks is an effective approach to ensuring that a mobile robot can remain responsive to sudden and unpredictable changes in the environment. RTAI provides an ability to make some processes as “more important” by moving time-critical routines out of the Linux environment and into an environment managed by a real-time scheduler. By

running Linux *on-top-of* RTAI, the real-time scheduler runs the entire Linux environment as the idle process, maintaining the ability to preempt it when real-time processes become ready to run. The RTAI hardware abstraction layer intercepts and saves interrupt signals destined for the Linux kernel until all real-time processes have completed, when Linux is the running process RTAI passes the missed interrupts to the kernel, making Linux largely unaware that it is being subverted by the real-time scheduler [8].

The RTAI microkernel makes the following real-time services natively available to the developer: The LXRT package allows applications to dynamically designate POSIX threads as hard real-time tasks. Concurrency library supports priority inheritance, supplying read/write locks and semaphores that detect and avoid deadlock due to priority inversion. Precision clock allows developers to set timers and instrument processes with nanosecond granularity ($10^{-9}$ seconds).

Despite the added complexity of working with a real-time microkernel, the services afforded to the developer simplify the creation of systems that maintain consistent periodic execution schedules as a means of detecting and responding to a changing environment. The simplicity of this approach is that unbounded processing loads are allowed within the Linux user-space because the time-critical routines are managed by a real-time scheduler capable of preempting the entire Linux environment within a fixed period of time.

The next logical question is, “How many real-time tasks can be supported by this approach?” Like YARA, this experiment focuses on allowing low-level control routines to remain predictably responsive to changes in the environment while sharing a single processing resource with computationally intensive routines. Although isolated from the effects of unpredictable fluctuations in a system’s computational load, the ability of a system to remain predictably responsive requires that the real-time domain behaviors identified as time-critical be managed as real-time components and do not jeopardize the system’s operational requirements. In other words, the determination of which, how many, and the timing constraints associated with the development of a real-time behavior based architecture are going to be dependent on the domain requirements.

## *5.4 Summary*

The results obtained in this chapter demonstrate the ability of the UBF's modular design to simplify development and testing by keeping behaviors focused, supporting code reuse, allowing large hierarchical designs and encouraging experimentation. Additionally, the modular design of the framework allows elemental behaviors to be implemented as periodic tasks in a real-time operating system as a means of ensuring the responsiveness of critical routines. The first experiment, Case Study I, demonstrates how the modular design encourages experimentation through rapid prototyping and testing. Based on the elemental behaviors used in the experiment, fusion based arbiters, which arbitrate on a per motor command basis, provide more robust outward behaviors than traditional winner-take-all selection approaches. The second experiment, Case Study II, capitalizes on the framework's ability to compose behavior structures and applies a genetic program as a means of automating the discovery of good behavior structures for domains where the optimal solution is unknown. The use of a stochastic search is able to discover effective solutions for homeostatic and multi-objective domains that are up to 122 percent better than one created by an expert. The final experiment, Case Study III, demonstrates the ability of the framework to be used in a real-time context, allowing reactive and deliberative tasks to be interleaved while ensuring safe, dependable robot operation by guaranteeing that low-level control routines remain predictably responsive. This experiment demonstrates the ability of a periodic control routine to become the running process in less than 100 μs independent of the system's computational load.

# VI. Conclusions

This investigation demonstrates that behavior-based control architectures share critical aspects and can be represented by a single straightforward unified behavior framework (UBF) and that the use of real-time scheduling technologies can allow low-level control routines to operate at scheduled intervals that are predictably responsive. This chapter reiterates the need for structured approaches towards software development and the need to incorporate real-time scheduling technologies to establish a responsive basis of low-level control for mobile robots. The test results from Chapter V are summarized in section 6.2, and are followed by possible areas for future work. The final section, 6.4, presents the final remarks of this thesis.

## *6.1 Summary*

The development of the UBF is intended to provide a framework for the development of reactive behavior architectures and is a structural guide that applies standard software engineering approaches to simplify development and testing of mobile robot controllers. Additionally, the modular design of the UBF allows the base behaviors to be implemented as real-time tasks to ensure the responsiveness of low-level control routines that are time sensitive or contribute to the safe operation of the system.

Traditionally, a mobile robot design implements a single behavior architecture, which binds its performance to the strengths and weaknesses of that architecture. Monolithic implementations are further limited because they are platform specific and not reusable between robots. Instead of pursuing this, the UBF makes a separation between the controller and the reactive behavior logic. In order to do this, a strategy pattern establishes a family of interchangeable behaviors. The UBF also addresses the need for scalability by providing construction tools that allow robust structures to be formed as arbitrated hierarchies of small, highly focused components. The use of the composite pattern ensures that the resulting structures are scaleable and belong to the established family of behaviors. This approach eases design complexity, allowing atomic behaviors to be designed, implemented and tested independently and then joined together

to produce rich and coherent behaviors. The ease with which components can be formed into stable structures encourages reuse and experimentation.

Driven by the requirement that an autonomous vehicle must not only be responsive in dynamic environments but rational and deliberative as well, the three-layer architecture has become a common paradigm for designing autonomous robot control architectures. Each layer of the architecture executes concurrently with the others, each pursuing their respective goals.

The establishment of a family of behavior algorithms that can be used interchangeably by a robot's low-level controller allows for the system to change its active behavior at runtime and provide a responsive and flexible basis of control for implementation in a three-layer architecture. This approach also gives a developer the freedom to use the behavior-based system they feel is the most appropriate for the given domain. The ability to develop focused behaviors as modules eases the complexity of designing and testing new behaviors. Such atomic behaviors can then be combined into arbitrated hierarchies that produce behaviors that are robust at the highest level.

Despite the use of concurrent programming techniques, current three-layer architecture implementations are unable to guarantee that their reactive control processes will execute at regular intervals. This is not a failing of the architecture, but a failure of the thread scheduling algorithm used by modern operating systems. Applications are emerging where responsiveness is important and milliseconds of delay matter, and it is no longer enough to say that the highest priority process will be the next process to run. Instead, real-time tasks require a guarantee that the highest priority process will become the running process in set period of time. Case study III presents the UBF in a goal directed configuration performed using a Pioneer P2-AT8 robot running Linux on top of RTAI, and an adaptation of the Player control suite. By treating the goal directed behavior as a time critical task the system maintains a stable basis of reactive-control that becomes the running process in less than 100 μs, regardless of the current process load.

## *6.2 Results*

The results obtained in this thesis demonstrate the ability of the UBF's modular design to simplify development and testing by supporting code reuse, large hierarchical

designs and experimentation. Additionally, the modular design of the framework allows elemental behaviors to be implemented as periodic tasks in a real-time operating system. The first experiment, Case Study I, demonstrates how the modular design encourages experimentation through rapid prototyping and testing. Based on the elemental behaviors used in the experiment, arbiters which arbitrate on a per motor command basis provide more robust outward behaviors than traditional winner-take-all selection approaches. The second experiment, Case Study II, capitalizes on the framework's ability to compose behavior structures and applies a genetic program as a means of automating the discovery of good behavior structures for domains where the optimal solution is unknown. The use of a stochastic search is able to discover effective solutions for homeostatic and multi-objective domains that are up to 122 percent better than that of an expert. The final experiment, Case Study III, demonstrates the ability of the framework to be used in a real-time context, allowing reactive and deliberative tasks to be interleaved while ensuring safe, dependable robot operation by guaranteeing that low-level control routines remain predictably responsive. This experiment demonstrates the ability of a periodic control routine to become the running process in less than 100 μs regardless of the current computational load.

## *6.3 Future Investigation*

Case study III demonstrates the ability of low-level control routines to execute at predictable intervals despite intensive processing loads at higher levels. This approach is useful in that it allows computationally intensive deliberative processes to share processing resources by running "in between" reactive control routines without degrading the responsiveness of reactive control routines. This study uses fixed periodic intervals, however, ultimately a developer would like to have an ability to dynamically reschedule low-level processes to allow their periodic execution rates to adjust based on the level of change in the environment. Consider a routine that processes the current global positioning satellite (GPS) signals to calculate the robot's current position. If the robot is stopped or is moving slowly, the execution rate for this routine can be reduced, thus lending processing power to higher order routines. Conversely, when the robot is moving quickly this routine runs more frequently, reducing the amount of computation time

available to higher order processes. When used with a large base of reactive behaviors, an ability to expand and contract the execution rate of reactive tasks effectively allows mobile robot architectures to change from more deliberative to more reactive and back as the environment changes. The question that emerges is, “How can the rate of change in an environment be quantified and associated with the real-time execution rate of reactive control behaviors?”

## *6.4 Final Remarks*

Efforts to develop autonomous vehicles that reduce the need for human-in-the-loop control are emerging in domains and range from the exploration of Mars via autonomous planetary rovers, airplanes, and balloons to reconnaissance operations by military and law enforcement agencies using handheld UAVs (*unmanned aerial vehicles*). For systems operating in the real-world, an ability to detect and handle external events is paramount to providing safe and dependable operation, because their unexpected operation can affect lives and property in the real-world. The unified behavior framework presented draws on modern software engineering principles to simplify development of reactive controllers without locking a system developer into using any single behavior system. Additionally, the unified behavior framework, coupled with a real-time process scheduler, allows routines that are critical to a vehicle’s safe operation to be predictably responsive as well.

# *Bibliography*


[1] J. Aas. "Understanding the Linux 2.6. 8.1 CPU Scheduler." *Retrieved Oct*, vol. 16, 2005.

[2] R. C. Arkin. "Behavior-based robot navigation for extended domains." *Adaptive Behavior*, vol. 1, pp. 201-225, 1992.

[3] R. C. Arkin. "Survivable robotics systems: reactive and homeostatic control." *Robotics and Remote Systems for Hazardous Environments*, pp. 135-154, 1993.

[4] R. C. Arkin. *Behavior-Based Robotics*. Cambridge, MA: MIT Press, 1998.

[5] J. E. Baker. "Reducing bias and inefficiency in the selection algorithm." *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and their Application*, pp. 14-21, 1987.

[6] M. Barabanov, "A Linux-Based Real-Time Operating System," New Mexico Institute of Mining and Technology, 1997.

[7] M. Barabanov and V. Yodaiken. "Introducing Real-Time Linux." *Linux Journal*, vol. 34, pp. 19-23, 1997.

[8] T. Bird. "Comparing Two Approaches to Real-Time Linux." *CTO of Lineo*, 2000.

[9] J. Bloch. *Effective Java: Programming Language Guide*: Addison-Wesley, 2001.

[10] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*: Addision-Wesley, 2000.

[11] R. P. Bonasso, R. J. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack. "Experiences with an Architecture for Intelligent Reactive Agents." *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 9, 1997.

[12] R. P. Bonasso and D. Kortenkamp. "Using a Layered Control Architecture to Alleviate Planning with Incomplete Information." *Proceedings of the AAAI Spring Symposium\ Planning with Incomplete Information for Robot Problems*, pp. 1-4, 1996.

[13] R. P. Bonasso, D. Kortenkamp, and T. Whitney. "Using a Robot Control Architecture to automate Space Shuttle Operations." *Proc. of the 1997 National Conference on Artificial Intelligence*, pp. 949–956, 1997.

[14] V. Braitenberg. *Vehicles: Experiments in Synthetic Psychology*. Cambridge, MA: MIT Press, 1984.

[15] R. A. Brooks. "A Robust Layered Control System for a Mobile Robot." *IEEE Journal of Robotics and Automation*, vol. RA-2, pp. 14-23, 1986.

[16] R. A. Brooks. "Elephants Don't Play Chess." *Robotics and Autonomous Systems*, vol. 6, pp. 315, 1990.

[17] R. A. Brooks. "New Approaches to Robotics." *Science*, vol. 253, pp. 1227-1232, 1991.

[18] S. Caselli, F. Monica, and M. Reggiani. "YARA: A Software Framework Enhancing Service Robot Dependability." *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference*, pp. 1970- 1976, 2005.

[19] C. Coello Coello, D. Van Veldhuizen, and G. Lamont. *Evolutionary Algorithms for Solving Multi-Objective Problems*. New York, NY: Kluwer Academic, 2002.

[20] J. Connell. "A Behavior-Based Arm Controller." *IEEE Transactions on Robotics and Automation*, vol. 5, pp. 784-791, 1989.
[21] S. Enderle, H. Utz, S. Sablatnog, G. Kraetzschmar, and G. Palm. "Miro: Middleware for Autonomous Mobile Robots." *Robotics and Automation, IEEE Transactions*, vol. 18, pp. 493- 497, 2002.
[22] R. J. Firby, "Adaptive execution in complex dynamic worlds," Ph.D. Dissertation, Yale University, YALEU/CSD/RR #672, 1989.
[23] R. J. Firby. "Task Networks for Controlling Continuous Processes." *Proceedings of the Second International Conference on AI Planning Systems*, 1994.
[24] M. Fujita and K. Kageyama. "An Open Architecture for Robot Entertainment." *Proceedings from the First International Conference on Autonomous Agents*, pp. 435-442, 1997.
[25] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns*. Boston, MA: Addison-Wesley, 1994.
[26] E. Gat. "On Three-Layer Architectures." *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, pp. 195 - 210, 1998.
[27] B. Gerkey, R. T. Vaughan, and A. Howard. "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems." *Proceedings of the 11th International Conference on Advanced Robotics*, pp. 317–323, 2003.
[28] R. L. Glass. *Facts and Fallacies of Software Engineering*: Addison-Wesley Boston, 2003.
[29] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison-Wesley, 1989.
[30] I. Horswill. "A Laboratory Course in Behavior-Based Robotics." *IEEE Intelligent Systems*, vol. 15:6, pp. 16-21, 2000.
[31] P. Husbands. "Genetic Algorithm in Optimization and Adaptation." *Advances in Parallel Algorithms*, pp. 227 - 276, 1992.
[32] IEEE. *Portable Operating System Interface (POSIX)*: IEEE/ANSI Std 1003.1, 1996.
[33] L. P. Kaelbling, "An Architecture for Intelligent Reactive Systems," in *SRI International Technical Note No. 400*. Menlo Park, CA, 1986.
[34] K. Konolige. "The Saphira Architecture: A Design for Autonomy." *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 9, pp. 215-235, 1997.
[35] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press, 1992.
[36] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: MIT Press, 1998.
[37] P. Maes. "Situated Agents Can Have Goals." *Robotics and Autonomous Systems*, vol. 6, pp. 49 - 70, 1990.
[38] P. Mantegazza, E. L. Dozio, and S. Papacharalambous. "RTAI: Real Time Application Interface." *Linux Journal*, vol. 2000, 2000.
[39] H. P. Moravec. *Robot rover visual navigation*: UMI Research Press Ann Arbor, Mich, 1981.
[40] M. Nelson. "Robocode Central." *http://robocode.sourceforge.net*, 2006.
[41] N. J. Nilsson. *Shakey the Robot*: SRI International, 1984.

[42] H. Nyquist. "Certain Topics in Telegraph Transmission Theory." *Proceedings of the IEEE*, vol. 90, 2002.
[43] J. Rosenblatt. "DAMN: A distributed Architecture for Mobile Navigation." *the AAAI Spring Symposium on Lessons Learned for Implemented Software Architectures for Physical Agents*, pp. 167 - 178, 1995.
[44] J. Rosenblatt. "Utility Fusion: Map-Based Planning in a Behavior-Based System." *Field and Service Robotics*, pp. 411 -418, 1998.
[45] P. Sarolahti, "Real-Time Application Interface," Technical Report, University of Helsinki, Dept. of Comp. Sci. 2001.
[46] C. Schlegel. "Communicaions patterns for OROCOS. Hints, remarks, specifications." *Technical Report, Research Institute for Applied Knoledge Processing (FAW)*, 2002.
[47] C. Schlegel and R. Worz. "The software framework SMARTSOFT for implementing sensorimotor systems." *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'99)*, pp. 1610 - 1616, 1999.
[48] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Dicipline*. Upper Saddle River, NJ: Prentice Hall, 1996.
[49] M. Shuttleworth. "Ubuntu: Linux for human beings." *http://www.ubuntu.com/*, 2006.
[50] W. Stallings. *Operating Systems*, Third ed. Upper Saddle River, New Jersey 07458: Prentice Hall, 1998.
[51] P. Stevens and R. Pooley. *Using Uml: Software Engineering with Objects and Components*: Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.
[52] H. Utz, G. Kraetzschmar, G. Mayer, and G. Palm. "Hierarchical Behavior Organization." *2005 International Conference on Intelligent Robots and Systems*, 2005.
[53] M. Veloso, J. Carbonell, A. Perez, D. Borrajo, E. Fink, and J. Blythe. "Integrating Planning and Learning: The PRODIGY Architecture." *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 7, pp. 81-120, 1995.
[54] A. J. Wellings. *Concurrent and Real-Time Programming in Java*. West Sussex PO19 8SQ, England: John Wiley & Sons, Ltd, 2004.
[55] B. Woolley and G. Peterson, "Genetic Evolution of Hierarchical Behavior Structures," Technical Report, Air Force Institute of Technology, WPAFB, OH, 2007.
[56] B. Woolley and G. Peterson, "Unified Behavior Framework for Reactive Robot Control," Technical Report, Air Force Institute of Technology, WPAFB, OH, 2007.
[57] K. Yaghmour. "The Real-Time Application Interface." *Proceedings of the Linux Symposium, July*, 2001.
[58] V. Yodaiken and M. Barabanov. "A Real-Time Linux." *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*, January 1997.
[59] Y. Yokote. "The Apertos Reflective Operating System: The Concept and Its Implementation." *ACM SIGPLAN Notices*, vol. 27, pp. 414-434, 1992.

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 074-0188*



**PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

| **1. REPORT DATE** *(DD-MM-YYYY)* | **2. REPORT TYPE** | **3. DATES COVERED** *(From – To)* |
|---|---|---|
| 22-03-2007 | **Master's Thesis** | Aug 2005 – Mar 2007 |

**4. TITLE AND SUBTITLE**

Unified Behavior Framework for Reactive Robot Control in Real-Time Systems

**5a. CONTRACT NUMBER**

**5b. GRANT NUMBER**

**5c. PROGRAM ELEMENT NUMBER**

**6. AUTHOR(S)**

Woolley, Brian, G., First Lieutenant, USAF

**5d. PROJECT NUMBER**
06SN02COR

**5e. TASK NUMBER**

**5f. WORK UNIT NUMBER**

**7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)**

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

**8. PERFORMING ORGANIZATION REPORT NUMBER**

AFIT/GCS/ENG/07-11

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

AFRL/SNRN
Bldg 620, Room 3AJ39
2241 Avionics Circle
WPAFB OH 45433-7333 DSN: 785-6127 ext 4274

**10. SPONSOR/MONITOR'S ACRONYM(S)**

**11. SPONSOR/MONITOR'S REPORT NUMBER(S)**

**12. DISTRIBUTION/AVAILABILITY STATEMENT**

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**

Endeavors in mobile robotics focus on developing autonomous vehicles that operate in dynamic and uncertain environments. By reducing the need for human-in-the-loop control, unmanned vehicles are utilized to achieve tasks considered dull or dangerous by humans. Because unexpected latency can adversely affect the quality of an autonomous system's operations, which in turn can affect lives and property in the real-world, their ability to detect and handle external events is paramount to providing safe and dependable operation. Behavior-based systems form the basis of autonomous control for many robots. This thesis presents the unified behavior framework, a new and novel approach which incorporates the critical ideas and concepts of the existing reactive controllers in an effort to simplify development without locking the system developer into using any single behavior system. The modular design of the framework is based on modern software engineering principles and only specifies a functional interface for components, leaving the implementation details to the developers. In addition to its use of industry standard techniques in the design of reactive controllers, the unified behavior framework guarantees the responsiveness of routines that are critical to the vehicle's safe operation by allowing individual behaviors to be scheduled by a real-time process controller. The experiments in this thesis demonstrate the ability of the framework to: 1) interchange behavioral components during execution to generate various global behavior attributes; 2) apply genetic programming techniques to automate the discovery of effective structures for a domain that are up to 122 percent better than those crafted by an expert; and 3) leverage real-time scheduling technologies to guarantee the responsiveness of time critical routines regardless of the system's computational load.

**15. SUBJECT TERMS**

Artificial Intelligence, Robotics, Real-Time, Software Engineering, Object Oriented Programming

| **16. SECURITY CLASSIFICATION OF:** | | | **17. LIMITATION OF ABSTRACT** | **18. NUMBER OF PAGES** | **19a. NAME OF RESPONSIBLE PERSON** |
|---|---|---|---|---|---|
| **REPORT** | **ABSTRACT** | **c. THIS PAGE** | | | Gilbert L. Peterson, (ENG) |
| **U** | **U** | **U** | **UU** | 111 | **19b. TELEPHONE NUMBER** *(Include area code)* (937) 255-3636, ext 4281; e-mail: Gilbert.Peterson@afit.edu |

**Standard Form 298 (Rev: 8-98)**
Prescribed by ANSI Std. Z39-18